# A Globally Distributed System for Job, Data, and Information Handling for High Energy Physics

Gabriele Garzoglio

November 7, 2005

FERMILAB-THESIS-2005-32

School of Computer Science,
Telecommunications and Information Systems,
DePaul University
Chicago, Illinois

**Thesis Committee:**
Ljubomir Perković, Chair
Radha Jagadeesan
Martin Kalin
Richard St. Denis

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy*

**Abstract**

The computing infrastructures of the modern high energy physics experiments need to address an unprecedented set of requirements. The collaborations consist of hundreds of members from dozens of institutions around the world and the computing power necessary to analyze the data produced surpasses already the capabilities of any single computing center. A software infrastructure capable of seamlessly integrating dozens of computing centers around the world, enabling computing for a large and dynamical group of users, is of fundamental importance for the production of scientific results. Such a computing infrastructure is called a computational grid.
The SAM-Grid offers a solution to these problems for CDF and DZero, two of the largest high energy physics experiments in the world, running at Fermilab. The SAM-Grid integrates standard grid middleware, such as Condor-G and the Globus Toolkit, with software developed at Fermilab, organizing the system in three major components: data handling, job handling, and information management. This dissertation presents the challenges and the solutions provided in such a computing infrastructure.

## Acknowledgments

Throughout the five years spent working on the SAM-Grid project, I never realized how many people have influenced my professional and personal life. Some of them are still good colleagues, some others have become friends, some have moved on, leaving their sign on the SAM-Grid code and on my memory. A simple page of "acknowledgments" cannot summarize how much they meant to me, but, on a dissertation, this is the space that I have to at least remember them.

First, I wish to thank my advisor, Dr. Ljubomir Perković. Thanks to his patience, dedication, and support, he taught me how a researcher thinks and writes about his work. I want to thank the members of the committee, Dr. Radha Jagadeesan, Dr. Martin Kalin, Dr. Rick St. Denis, for giving me their feedback on my research.

I want to mention the colleagues of the SAM-Grid team. Igor Terekhov, the initial leader of the project, taught me how to implement ideas in real life software and make them work. Together, we shared the adventure of an exciting project and weathered the storm during the difficult times. Andrew Baranovski contributed with challenging ideas, which he implemented in high-quality code. His analyses and observations helped us understand the dynamics of a complex system. Rod Walker, our contact at Imperial College, London, was our first collaborator, user, and external developer. Parag Mhashilkar helped in the early stages of the SAM-Grid system as a student and today keeps improving the system as a Fermilab colleague.

I want to thank the other members of the SAM-Grid team, in particular Sinisa Veseli, Lauri Carpenter, Robert Illingworth, Steve White, Valeria Bartsch, Stefan Stonjek. My gratitude goes also to the SAM-Grid, Runjob, and DZero management for their support and interest, in particular to Adam Lyon, Wyatt Merritt, Rick St. Denis, Lee Lueking, Iain Bertram, Gavin Davies, and Amber Boehnlein.

For the past 4 years, thanks to the involvement of Dr. Jae Yu, University of Texas at Arlington, the SAM-Grid team has been strengthened by a vital student program. I want to thank for their energy and enthu-

siasm Siddharth Patil, Abhishek Rana, Hannu Koutaniemi, Vijay Murthi, Sankalp Jain, Aditya Nishandar, Annop Rajendra, Bimal Balan, and Sudhamsh Reddy.

The Condor Team, University of Wisconsin at Madison, has given a fundamental contribution in shaping the design and implementation of the SAM-Grid. I wish to thank in particular Todd Tannenbaum, Alain Roy, and Peter Couvares.

My gratitude goes especially to our user community. Without their feedback, we could have never developed a usable system, now deployed at dozens of sites throughout America, Europe, and Asia. I want to remember in particular Joel Snow, Daniel Wicke, Tibor Kurca, Mike Diesburg, Patrice Lebrun, Yann Coadou, Dugan O'Neil, Frederic Villeneuve, Sabah Salih, Joerg Meyer, Cano Ay, and Vlasta Hynek.

A personal thank you goes to my Fermilab line management, Margaret Votava, Ruth Pordes, Eileen Berman, and Don Petravick, who supported my decision of engaging in a graduate program, trusting that I would still give my best on my day to day job responsibilities.

Finally, I want to thank my family and my fiancée, Maura, for mitigating the stress of such busy years with their love and support.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The collaborations of high energy physics experiments are traditionally large and geographically distributed. Only a few laboratories around the world offer the facilities necessary to conduct innovative programs of research in the field. Physicists from all over the world gather at these sites, building particle detectors specifically designed to best exploit the unique characteristics of these facilities. Nowadays, collaborations involving scientists from dozens of different nationalities are becoming increasingly common. In addition, in the past decades, the complexity and the cost of these detectors have increased dramatically, as new and more advanced physics questions have become the focus of the community. To confront the unprecedented budgets and the challenges posed by the research programs, the collaborations have naturally increased in size, involving typically many hundreds of scientists.

The complexity of the physics and the detectors does not only entail larger numbers of collaborators. The amount of data to be gathered, catalogued, processed, and analyzed, in fact, is on the order of a Petabyte per year per experiment. Various factors contribute to the exponential growth of the data size. First, the observation of the physics of interest requires a much greater spatial resolution than in the past: the typical number of data channels of a modern detector reaches in the millions, an order of magnitude larger than a decade ago. Second, high energy physics studies phenomena

statistical in nature and most of the particles studied in today's research programs have an extremely rare probability of occurring. The characteristics of such rare particles can be studied with the required level of statistical significance only if enough of them are observed during the data taking. In the modern experiments, it is usual to gather data for many years and observe a few dozen occurrences of a certain rare particle, while the total number of background particles amounts to many billions.

In this scenario, addressing the computing needs of the modern high energy physics collaborations presents many challenges, and the amount of data and large geographical scale are only the most evident. For most experiments today, the processing power required to produce innovative scientific results is already larger than what a single central computing facility can provide. In the past, the multinational institutions participating in the experiment generally financed the hosting laboratory to buy the computing hardware and take the responsibility of managing it. Today, this trend is reversing, and, typically, national funding agencies are willing to invest in computing infrastructures as long as they are maintained locally and dedicated to more than one experiment or even discipline. This politics is popular because it promotes the development of national computing centers and expertise, and, at the same time, it streamlines the maintenance of the computing systems by promoting resource sharing. This trend leads to the formation of computing centers very diverse with respect to hardware, software systems, configurations, management policies, and availability. A global software infrastructure capable of interfacing to each individual center is therefore the key to enabling transparent access to the total pool of computing resources.

Besides the diversity of the computing fabric, another important challenge is the dynamic nature of the membership to the collaborations. Typically, scientists join and leave the physics collaboration throughout the lifetime of the research program. This group of people, temporarily working together to accomplish a common goal, is sometimes referred to as a "Virtual Organization". A successful distributed computing infrastructure must preserve the efficiency and, at the same time, guarantee the security of the

computing environment.

In summary, the broad question that this research addresses is:

How do we implement, deploy and operate a global software infrastructure that enables a Virtual Organization to handle Petabytes of data and to access a Petaflop-scale pool of shared and distributively owned resources, in a secure, accountable and transparent fashion?

This software infrastructure is called a "Computational Grid" [1], in analogy to the Electrical Power Grid: in the same fashion as the power grid provides electricity to the users irrespectively of where it is produced, analogously, the computational grid gives transparent access to geographically distributed computing resources.

## 1.1 The Standard Grid Middleware

Astronomers and high energy physicists have started moving toward grid computing between five to ten years ago (as exemplified by current physics projects such as BaBar [2], Belle [3], DZero [4], CDF [5], SDSS [6], and LIGO [7]). Many groups of physicists and computer scientists are also building a large grid for the next generation of experiments, which will start taking data in 2007 at the Large Hadron Collider (LHC) at the European Laboratory for Particle Physics (CERN) [8], Switzerland (see the collaborations of CMS [9, 10], Alice [11, 12] , LHCb [13], and Atlas [14]). In this scenario, the proliferation of distinct solutions to very similar problems risks to become a big concern. Not only is this a duplication of effort in a time when the budget for high energy physics research grows by only one to two percent per year, but this also implies in projection a cost to educate the scientists to use many different computing infrastructures. At this point, there is clearly a need for the scientific community to come together and define what services a grid must provide and by what protocols and interfaces.

Since 2001, these matters are discussed by hundreds of people gathering at the Global Grid Forum (GGF) Conferences [15]. The conferences are organized around working groups, which provide documents to define the

specific problems, to propose the standards and, sometimes, provide reference implementations. The leader in this effort is the Globus Alliance [16], which provides open source technologies that are used as building blocks of many grids for scientific communities and the industry. Their Globus Toolkit [17] provides a *de facto* standard implementation of four major components of a grid computing system:

1. **Security**: the Globus Security Infrastructure (GSI) is based on a Public Key Infrastructure that uses X509 certificates. Certificates define the identity of software services and people. They are generated by users and administrators and certified (signed) by Certificate Authorities, which, ultimately, define the trust relationship in the system. All the software in the Toolkit is integrated with GSI.

2. **Data Management**: these components offer data movement via a GSI-enabled FTP service called GridFTP, and replica location services via the Globus Replica Catalog.

3. **Information Services**: the resources of a computing cluster and the jobs running therein can be monitored using the Monitoring and Discovery Service (MDS), a GSI-enabled version of an LDAP server. MDS offers also an indexing service, which provides access to information across multiple information servers.

4. **Resource Management**: provides a reference implementation of the Globus Resource Allocation and Management (GRAM) protocol. The GRAM server, called gatekeeper, offers simple interfaces to manage jobs running on underlying batch systems.

Various other groups also provide similar or competing software components, which are less popular than the Globus Toolkit within the high energy physics community (see Avaki [18], Platform Computing [19], Entropia [20], Sun Grid Engine [21], United Devices [22], Parabon [23], ProcessTree [24], Popular Power [25], Mojo Nation [26], and DataSynapse [27]).

Despite its wide acceptance, the components of the Globus Toolkit are

too low-level to be usable "out of the box" as a complete grid solution. The main limitations of the GRAM protocol are:

- The job instance is not persistent: in case of a crash of key remote machines, the system looses the job request. The Globus Toolkit does not implement reliable job management.
- There are no provisions for error recovery mechanisms: in case of fault conditions in the infrastructure, the system is incapable of reacting. The Globus Toolkit is not fault tolerant.
- The user interface is not user friendly: the Globus Resource Specification Language (RSL) is arguably complex, verbose, and error prone.

The limitations of the Globus Toolkit's reference implementations of the GRAM protocol are discussed further in chapter 4.

The Monitoring and Discovery Service is well suited for gathering information upon request, but:

- It does not allow information to be pushed into the system, a characteristic needed to monitor event driven systems. MDS implements only a pull-based monitoring paradigm.
- The reference implementation is generally considered low quality (sec. 2.2.2).

The Data management tools provide only a limited number of components, most of which have been implemented only recently:

- The data movement component, called GridFTP, was the more robust and mature service, being based on the WU FTP implementation of the transfer protocol. In recent software releases, because of licensing issues, the server has been completely reimplemented, including the stable and trusted security module. The other components available only in the more recent releases of the Globus Toolkit are the replica location and the reliable file transfer components.

- Real-life implementations of data management services, such as data handling or storage systems, require components that are not available in the toolkit. Examples of these components are space allocation, data caching, and scratch space management systems.

Because of these limits, various groups have developed software that offer higher-level services, building on top of the Globus Toolkit [28, 29, 30, 31]. A leading such group is the Condor Team, the developers of the Condor batch system [32]. For more than a decade, they have been developing software to support high throughput computing, delivering large amounts of processing capacity over long periods. In the past years, they have reused some of the concepts of the Condor batch system to address the problem of job management on the grid. Condor-G [33], the GRAM enabled extension to Condor, provides reliable and robust job management via durable distributed transactions and error recovery mechanisms as well as a user-friendly system interface.

The Globus Toolkit and Condor are solutions so widely used together that are often referred to as the standard middleware. They are the central piece of various middleware distribution packages [34, 35] and are the basic components of many grid projects throughout the world [36, 37, 38, 39, 40, 41, 42, 43, 44, 45].

The general question that is the focus this research can therefore be rephrased as

How do we integrate the standard middleware to achieve a complete Job, Data and Information management infrastructure for the distribution, processing, and storage of physics experiment data?

and the corollary questions

What parts of the standard middleware need to be enhanced to address the typical huge scales of High Energy Physics? What parts are currently not sufficiently mature and must be replaced with in-house development?

We have been addressing these questions at Fermilab during the past 4 years in the context of the SAM-Grid project.

## 1.2 The SAM-Grid System

The SAM-Grid [46, 47, 48, 49, 50, 51] is a computing project started at the Fermi National Accelerator Laboratory [52] in January 2002 and it is one of the first grids deployed for the high energy physics community. The Laboratory, in Batavia, Illinois, operates the particle accelerator with the highest energy in the world, the Tevatron. The accelerator is used by two of the largest particle physics experiments currently taking data: CDF (Central Detector at Fermilab) [5] and DZero [4]. The project is conducted as a collaborative effort between physicists and computer scientists and it is financed in part by Fermilab, the Particle Physics Data Grid (PPDG) [53], in the US, and GridPP [54], in the UK. The goal of SAM-Grid is to enable fully distributed computing for the DZero and CDF experiments, integrating standard grid tools with in-house solutions and software when the standards do not provide an appropriate solution.

The SAM-Grid architecture is composed of three major components: the data handling, the job and the information management systems. This division is mostly natural as it closely follows the organization of the standard middleware and best capitalizes on the software already developed at Fermilab for the experiments. The most notable of this in-house software is the Sequential Access via Metadata (SAM) [55, 50, 56, 57, 58, 59, 60, 61, 62], the data handling system that we have developed for the experiments. Figure 1.1 and 1.2 shows two architectural diagrams of the SAM-Grid.

We now introduce the three major components of the SAM-Grid together with its division to global and local services. We also introduce the larger questions that have arisen during the phase of design, implementation, and deployment of each component. This dissertation will then address the questions related to the job management (chap. 3) and information management (chap. 2) components. We will also describe the interface between Grid services and the Fabric i.e. the ensemble of local resources and services at a site (chap. 4). In section 1.3 we summarize the solutions to the questions related to the data handling component, SAM, which we developed in an earlier project. We include this summary for completeness and because the

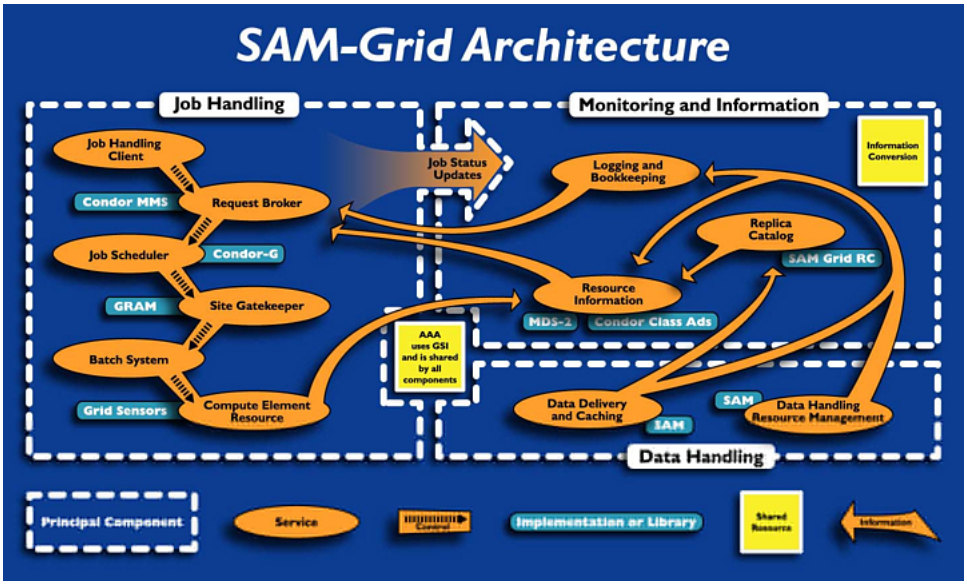


Figure 1.1: The SAM-Grid is divided into three major components: data handling, job handling and information management. All three components are integrated with strong security mechanisms. Each orange bubble represents an abstract aggregated service, whose implementation appears in the blue label next to it. The major challenge of the project was integrating all the services to enable globally distributed computing for the DZero and CDF experiments at Fermilab.

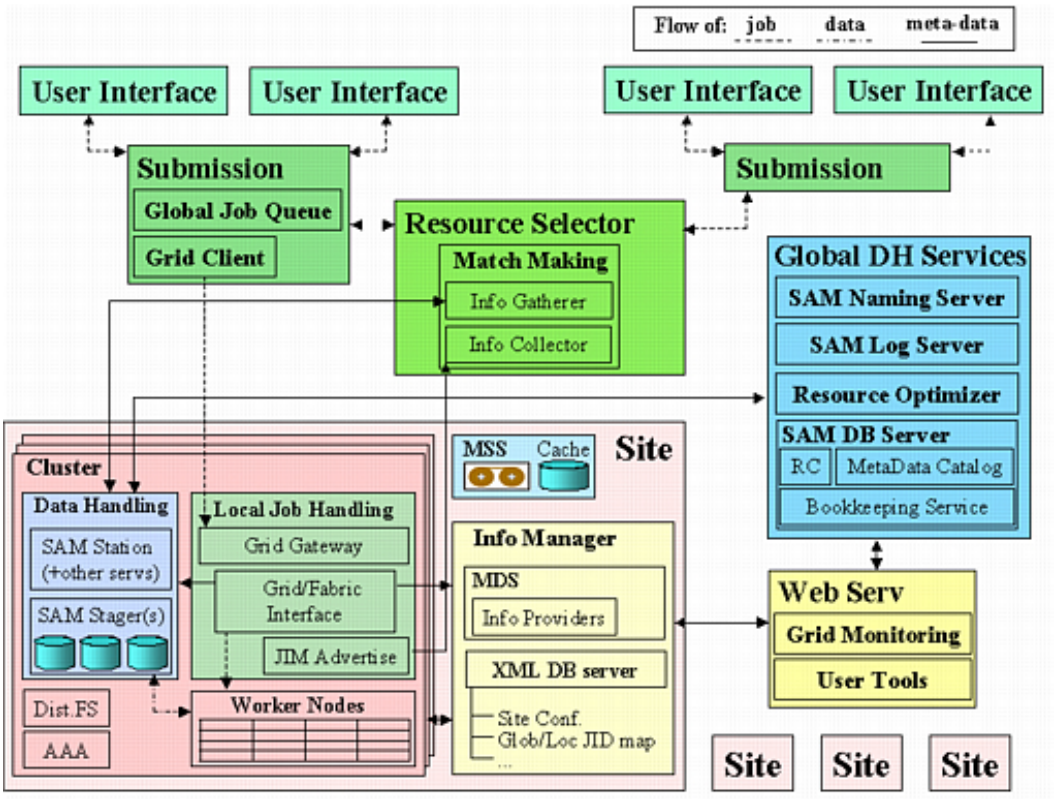


Figure 1.2: The SAM-Grid architecture can be decomposed in Fabric and Global services. Fabric services, local in nature, are shown in the pink boxes labeled "Site". Global services are represented by the remaining boxes. The diagram also shows the division of the SAM-Grid in three components. The Job Management, in green, is composed of a Local Job Handling service per site, a central Resource Selector, dozens of Submission services, and hundreds of User Interfaces. The Data Handling component, in blue, has Data Handling services at each site, and semi-central Global Data Handling Services. The Information Management, in yellow, is mainly constituted by Fabric services, interacting with information visualization mechanisms. Different types of arrows show the flow of jobs, data, and meta-data, tieing together services and components.

SAM system is of fundamental importance for both of job and information management.

As the SAM-Grid is expanding in size and usage, we are gathering useful first hand information on the challenges of deploying and operating a grid for high energy physics. In addition to describing the system, the dissertation presents such challenges and reports on the "lessons learned" for this first generation of grid systems.

### 1.2.1 Data Handling

The main problems that a Data Handling System should solve are summarized by the following four questions:

**How do we provide reliable data storage?** Whether coming out of a detector or out of a computing cluster after days of computation, data is ultimately the most import resource of all (carbon-based resources apart). A data handling system must provide a high probability of securing high-profile data to permanent storage.

**How do we enable global data distribution?** With hundreds of people at hundreds of collaborating institutions, a data handling system must provide reasonably easy and efficient access to the data throughout the globe, in order not to be the limiting factor to scientific discovery.

**How do we catalogue the data?** Organizing the data is a fundamental pre-requisite to data processing. This problem is non-trivial because the amount of data taken throughout the lifetime of a modern high energy physics experiment ranges between ten and twenty Petabytes, generally organized in tens of millions of files.

**How do we manage data handling resources?** With such a large amount of data, distributed throughout the world to increase accessibility, the data handling system controls dozens to hundreds of millions of dollars worth of equipment. Thus, the system represent an opportunity for the global coordination of hardware and software resources, in order to accomplish the goals of the physics program.

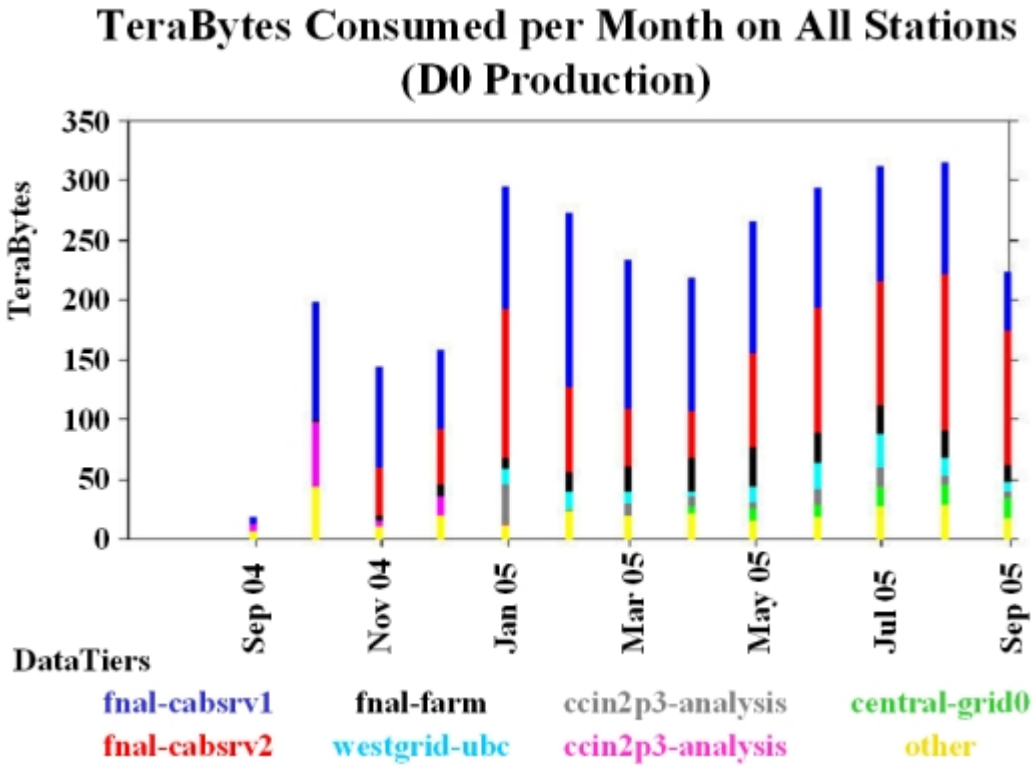


Figure 1.3: The amount of data consumed by DZero applications using SAM surpasses 300 Terabytes per month.

In the SAM-Grid, these questions are addressed by the SAM data handling system. The SAM project was started at Fermilab in 1997 to address the data handling challenge that the DZero experiment was going to face throughout the following decade. As the system grew more configurable and operationally stable, in 2001 CDF opted to adopt SAM for its data handling needs. Today the system manages a throughput of Terabytes of data per day throughout dozens of sites in America, Europe and Asia (fig. 1.3).

Other groups have developed systems that address some of the questions listed above (SRB [63], GDMP [64, 65], Giggle [66], NeST [67], Magda [68], DataCutter [69], and the Global Grid Forum [70]). As of today, SAM is still

arguably the most comprehensive data handling solution for the high energy physics domain.

The flexibility and stability of the SAM system is of central importance for the job and information management infrastructure of the SAM-Grid. Both services rely heavily on the maturity of SAM, in order to provide solutions to classical grid problems, such as data pre-staging and job/data co-location (sec. 3.4.1). For these reasons, the SAM system solutions are summarized in this dissertation (sec. 1.3), even if the focus of this research is the job and information management components.

### 1.2.2 Job Management

In high energy physics, as in other fields, a computational task can be decomposed in one or more units of computation, called jobs. During the process of design and development of the job management component, we have encountered a series of problems that a job management infrastructure must address. These problems are introduced hereby and discussed in detail in chapter 3.

**How do we enforce security?** A secure and accountable access to services and resources is the fundamental requirement to enforce any agreement between the experiments and the associated institutions.

**How do we handle the jobs reliably?** A grid is a dynamic environment in which resources and services do not have 100% availability. A job management infrastructure must treat the job instance persistently, handling the job to completion irrespectively of service and resource availability (sec. 3.3). It should be noted that job completion does not mean job success. In other words, job reliability does not entail fault tolerance.

**How do we avoid continuous network connectivity of the client?** The job management infrastructure should not require high availability of the machine where the user develops his/her applications. The client machines are, in fact, often laptops. This requirement naturally introduces a multi-tiered approach to job management(sec. 3.3).

**How do we automatically select resources?** The job management infrastructure should provide a service responsible for the selection of the resources on behalf of the users (sec. 3.4). This service should shield the user from the knowledge of: (1) the details of the grid resources; (2) the "best" resource to run a job.

**How do we run "structured" jobs?** Jobs can be composed of multiple processing stages. We call a job "structured" if it exposes to the grid the details of its processing stages. Since structured jobs for high energy physics run for days or weeks, the ability of processing them automatically reduces the possibility of human errors and the time to completion (sec. 3.2.4).

**How do we run generic applications?** Ideally, a grid should offer a set of services with standard interfaces that the jobs and the services themselves could use to accomplish the computational task. In practice, most high energy physics jobs today are hardly portable and very seldom grid-aware. In order to compensate for this, it is the middleware that needs to be specialized to deal with such jobs, for example by triggering collective services on behalf of grid-unaware jobs. The specific services involved and their optimization depend on the "type" of job i.e. on the user application (sec. 3.2).

**How can we be fault tolerant?** A reliable job management infrastructure handles the jobs to completion even in case of unavailability of key services. Because of services outages, though, a potentially successful job can fail. A fault tolerant system reacts to the infrastructure-related job failures, increasing the probability of success for the job (sec. 3.3).

**How do we meet the performance requirements?** Grid infrastructures are naturally less efficient than job management systems specialized for local hardware and software configurations. On the other hand, the advantages of a grid infrastructure, such as software standardization and transparent access to the resources, make the grid system convenient and cost effective. In addition, the inefficiencies typical of a grid system can be mitigated by giving grid services knowledge of the applications that use the resources (sec. 3.2).

**How do we scale with the number of jobs, users, resources, services?** The SAM-Grid infrastructure has proven to scale up to dozens of execution and submission sites and thousands of machines. We can run hundreds of concurrent jobs at a site and hundreds to thousands overall (sec. 5.1). Our current user-base is in the dozens of users and we envision the need to scale up to the hundreds.

The Condor and the SAM-Grid teams decided in 2001 to address the computational requirements of large high energy physics experiments by enhancing the Condor-G framework [71]. The advantages of such approach consist mainly in promoting the standards and ease the maintenance of the infrastructure by encapsulating the domain specific software to a series of modular, lightweight plug-ins.

We have found that the first three of the problems mentioned above were addressed in the standard middleware or could be addressed with minor modifications to the infrastructure. In order to enforce security, in fact, the SAM-Grid adopts the Globus Security Infrastructure as the basis of its security model. Reliability of job handling, instead, is inherent in the SAM-Grid via the reliability mechanisms that the Condor-G framework implements on top of the Globus Resource Allocation and Management (GRAM) protocol. Finally, continuous network connectivity is not required as the SAM-Grid implements a lightweight client environment based on the Condor-G client software. On the other hand, for the last problem, the current implementation addresses the needs of our community and we are investing minimal resources in investigating the problems further. The solution to the problems introduced by the remaining questions form the basis of this research for what concerns job management.

The job handling component of the SAM-Grid is designed to provide robust and efficient management of jobs on the grid. In addition, it provides a user interface specifically designed for high energy physics applications. Various groups address this problem in the context of other grid infrastructures. The European Data Grid (EDG) [36] software has provided a solution with the development of a large, arguably monolithic code base, based on

the standard middleware. LCG [37] is extending the infrastructure of EDG to address the needs of the LHC experiments. The GriPhyN (Grid Physics Network) [42] project uses the standard middleware for the low level job management, and introduces the concept of virtual data to organize job dependencies. See also the DIRAC [72] and GANGA [73] projects.

### 1.2.3 Information Management

The main problems that an information management infrastructure must address are:

**How do we do the bookkeeping of the data processing activity?** For high energy physics in particular, a solution to this problem enables the reproducibility of the physics results. Such reproducibility is fundamental for the accountability of the scientific discovery (sec. 1.3).

**How do we monitor jobs, services, and resource usage?** An infrastructure capable of gathering the status and history of main entities of the grid is useful for both users and other grid services. It is useful for users because it helps tracking the progress of their jobs and the status of the resources. It is useful for other services, such as resource selectors, for the discovery of the resources and the propagation of the resource characteristics (sec. 2.2).

**How do we configure distributed resources and services at a site?** A uniform framework to address configuration management is the key to the consistency of the information propagated through the grid (sec. 2.1).

The SAM-Grid relies on the SAM data handling system for data processing bookkeeping (sec. 1.3). For the problem of resource discovery, instead, the SAM-Grid resource selection service is based on the information collection mechanisms of the Condor match making service (sec. 3.4.2). Other grids also base their resource selection mechanisms on their information services. The resource selection service of LCG, also called the broker [74] relies

on the R-GMA (Relational Grid Monitoring Architecture) [75] information service to function.

The SAM-Grid information management infrastructure is discussed in chapter 2. In the SAM-Grid, the information management is organized into three categories:

1. The static or semi-static information, which deals with the configuration of the grid products, services and resources (sec. 2.1).

2. The dynamic information, which is mostly used for monitoring (sec. 2.2).

3. The historical view of the previous information, which consists of the logging and bookkeeping services (sections 2.2 and 1.3).

Each category of information, when considered in the context of a grid system, presents a different set of challenges.

### 1.2.4 Fabric Services

In order to reliably execute jobs, the job management component of every grid depends on a set of services local to the computing site. In addition to standard services, such as a local scheduler, the grid needs services that manage the grid jobs and the resources at the site. These services are sometimes called the fabric services. Typically, they provide local batch system adaptation for the incoming grid jobs, dynamic product installation, intra cluster transport of the job and its output, etc. For most of these services, standard and mature technologies are not available to the community, and they thus need to be developed.

The main fabric-level problems of a global computing infrastructure are:

**How do we foster site autonomy?** "Classical" distributed systems are designed using an architecture that assumes that the system has full control over the resources. Distributed components typically communicate using proprietary interfaces. Even if there is general consensus over the choice

of the transport protocols, historically there is little effort in defining "standards" for the interfaces themselves. What sets apart the paradigms of grid and peer-to-peer distributed systems is the sharing of distributively owned resources. A grid site instantiates distributed components that offer *services* adhering to standard protocols and interfaces. Within this paradigm, a grid infrastructure must allow a site the freedom to choose the architecture of its resources, the implementation of its services, the definition of its policies, etc.

**How do we run on non-dedicated resources?** The software developed for resources dedicated to accomplish a certain task is generally highly tailored to the specific configuration of the system e.g. network topology, organization of the local storage, software environment, etc. A grid infrastructure, on the other hand, must be able to interface to a variety of system configurations, ideally imposing minimal constraints on such configurations (chap. 4).

**How do we coordinate fabric services to execute complex jobs?** Offering a set of services to a job is not enough to achieve the quality of service necessary to successfully execute complex jobs. Services need notification and coordination before a job attempts to access them (chap. 4).

**How can we overcome temporary failures of fabric services?** Robustness in the access to local services is necessary to achieve quality of service. Transient failures that are acceptable for interactive usage of a system have dire consequences in the job success rate, when the same system is accessed via the grid (sec. 4.2).

**How do we overcome the lack of fabric services at a site?** Some sites do not implement services that are important for the grid. Typical examples are scratch space management or ubiquitous access to storage. In each case we have implemented software to mitigate the lack of these services. Yet, we believe that our "mitigators" do not provide a solution general enough and we consider this still an open problem.

The SAM-Grid runs on non-dedicated resources by encapsulating the pe-

culiarity of each site in the configuration of a single gateway node. At this node, the use of an "active" grid-to-fabric interface achieves the coordination of the fabric services at the site. The main advantage of this approach is the optimization of information access, such as job and data co-location or aggregation of database queries. Transient failures of the fabric services are overcome by retrials in the communications with the services. For example, the introduction of retrials upon denial of service have increased the job success rate of the SAM-Grid from 60% to 95%. A discussion of the design and the implementation of the SAM-Grid fabric services can be found in chapter 4. Throughout the entire dissertation we will argue how the SAM-Grid fosters site autonomy, one of the fundamental paradigms of grid computing. We will also argue throughout the dissertation that making the grid aware of details of the applications enables the implementation of resource optimizations not easily achievable otherwise.

## 1.3 The SAM Data Handling System

The SAM (Sequential Access via Metadata) data handling system [55, 50, 56, 57, 58, 59, 60, 61, 62] was designed and implemented with four principal goals in mind:

1. Provide reliable data storage, either directly from the detector or from data processing facilities around the world.

2. Enable data distribution to and from all of the collaborating institutions, today on the order of 70 per experiment.

3. Thoroughly catalogue the data for content, provenance, status, location, processing history, user-defined datasets, etc.

4. Manage the distributed resources to optimize their usage and, ultimately, the data throughput, enforcing, at the same time, the policies of the experiments.

The SAM system achieves these goals via a global network of cooperating software services, some of which are centralized in nature, like, for exam-

ple, the metadata catalogue, while some are fully distributed. Among the distributed services, the principal and foremost is the SAM station, whose role is pooling together a set of resources at a site, such as storage, computing, and network systems, for the sake of data management. Storage systems resources are either mass storage systems (MSS) or volatile storage systems. An example of mass storage system particularly important to both experiments is Enstore, a grid-enabled robotic tape-based storage, located at Fermilab, where all of the primary and most of the derived data are permanently stored. The Enstore system presents the data as a virtual file system and can cooperate with a Disk Cache (DCache) front layer, thus enabling efficient network data access. In addition, the DCache system optimizes tape management, a responsibility that in the past was left to the data handling, mainly because of its ability to efficiently buffer data requests. However, that responsibility requires a very detailed knowledge of the organization of the MSS tape library. Thanks to the DCache layer, Enstore moves several Terabytes of data daily in and out of the storage. Volatile storage systems, on the other hand, are present at almost every site in the form of disks. These disk resources can be optionally managed directly by the SAM system, with various caching algorithms and allocation policies, such as, for example, fair share utilization policies of storage space for all physics groups.

The system achieves data storage and distribution by coordinating the access of the globally distributed SAM stations to globally distributed storage systems. In this regard, the primary and foremost service offered by the data handling system is data replication. The service is automatically triggered by clients unable to directly access the data, due to topological constraints. Using delegation among stations, SAM transfers the data to a storage system accessible by the client, cataloguing, at the same time, the intermediate data locations. The route decided for the transfer depends on the selection criteria of the initial replica and on the topology of the stations, parameters that can both be tuned to shape the network traffic. In order to achieve replication, SAM implicitly relies on its data movement service. For this service, data transfer is a layer of abstraction independent of

the protocols, which are implemented, in the end, through a plug-in mechanism. In addition, to overcome possible errors during data transfer, SAM implements reliability in the form of data integrity checks, automatic retrial mechanisms, and, most importantly, alternative replica selections.

To catalogue the data, it was decided in 1997 to rely on a central, well-maintained relational database, managed at Fermilab. Despite the fact that a central service represents a potential single point of failure, the SAM database has turned out to be year after year a very reliable component. The information stored in the database ranges from system configuration, such as available resources and users, to data description (raw detector data, derived data, binaries, etc.). A particular effort has been put into the design of a meta-data schema flexible enough to allow the usage of SAM by more than one experiment, by enabling custom defined parameters. In this framework, users can define datasets giving logical names to queries in the meta-data parameter space. In addition, custom defined parameters have the advantage of limiting the necessity of schema evolutions. Nevertheless, in order to further isolate the client code from possible changes in the schema and, at the same time, hide the complexity of the internal data representation, SAM maintains multiple instances of a middle-tier server, which mediates the accesses to the database. Another fundamental aspect of the schema is the data processing history, which, combined with the meta-data catalogue, provides a record of the data provenance, an aspect crucial for the reproducibility of the physics results.

# Chapter 2

# The Information Management Component

A software environment designed to support the computing of a modern high energy physics experiment needs to address the challenges associated with handling large amounts of data, processed by jobs that run on a distributively owned, shared and dynamic resource infrastructure. Information management can benefit from a classification of the information in three categories, treated in practice with different software tools:

1. the information that is static or semi-static in nature, such as the global parameters of the grid or the setup of services and resources at a site. Changing this information generally involves a human intervention, typically by a system or service administrator. We refer to this category of information with the generic name of grid configuration and we talk more about it in section 2.1;

2. the information associated with the behavior of the entities in the grid, such as resources, services, jobs, et cetera. This information is dynamic in nature and it is captured in various degrees and representations by the monitoring infrastructure of the grid (sec. 2.2);

3. the historical view of the previous two categories, with the appropriate level of synthesis: this category is useful mainly for bookkeeping. This

activity is of fundamental importance for the reproducibility of scientific results. A reproducible data analysis, in fact, is indispensable to convince peer reviewers of the soundness of the scientific publication and, in the end, the credibility of the collaboration. Bookkeeping is also useful for statistical studies of the grid. These studies are interesting for comparing theoretical models with physical systems and for optimizing the usage of services and resource. We discuss this last category of information in section 2.2.4.

The work and literature on monitoring systems is conspicuous [93, 94, 95, 91, 75, 100, 102, 101, 104, 106]. These and other references are discussed and compared with the SAM-Grid solution in section 2.2. On the other hand, the work on configuration management is not at the same level of maturity. In section 2.1, we present the problem and the SAM-Grid solution.

## 2.1 The Configuration Infrastructure

This section discusses the problem of configuring distributed resources and services at a site (sec. 1.2.3). To analyze the problem, we need to clarify in what context we talk about *configuration*. In particular, we focus on the configuration of those parameters that define the behavior of grid services. We do not attempt to offer tools to administer the whole grid infrastructure, such as operating system components (e.g. libraries and devices) or standard system applications (e.g. batch and file systems). This latter problem is addressed on the Grid by a few emerging software infrastructures [77, 78, 79, 80] that extend the classical software distribution mechanisms for local clusters. These tools address configuration at the level of the operating system and of a few standard applications. The configuration of high-level software services is in general more complex in nature and not necessarily configurable automatically. We describe the problem in more detail in section 2.1.1. The subsequent sections describe the SAM-Grid solution.

### 2.1.1 Problems in Configuration Management

There are various challenges that a grid configuration infrastructure must address, first of all the large number of parameters. This factor has two main impacts. The first is the high likelihood of name conflicts. In order to prevent name conflicts, the framework will have to use some form of name spacing for the parameters. The second impact is the potential for inefficient management of information. The framework must thus use a technology that optimizes access to the parameters. Another challenge is that the configuration should be distributed to foster site autonomy and maintainability, but, on the other hand, should be easy to manage from anywhere within the grid and resilient to concurrent management attempts. For example, in order to describe the configuration of different groups of resources and services at a site, the system should allow the organization of the information into a single repository, central to the site, as well as into multiple repositories managed by different administrators, with different access policies, et cetera. In any case, the specific organization strategy should be transparent to the information management mechanisms. In addition, it is crucial to present a consistent view of the configuration throughout the system, a task not trivial considering the distributed nature of the services and the fact that different services need to represent the configuration in different formats. For example, the resource advertisement service (see sec. 3.4.2) and the monitoring service must provide the same information regarding the resource characteristics of a certain site, even if they are physically instantiated on different machines. Figure 2.1 illustrates this concept with a diagram: a site provides a logically unique repository of information. Different services use this repository to give a consistent view of the site to the grid.

The requirements defined so far are useful to guide our design decisions, but, by themselves, are not enough to build a software system. In order to achieve this goal, the *question* of how to configure services and resources at a site must be better qualified to address real-life problems, such as (1) how the information is gathered and (2) how it is organized.

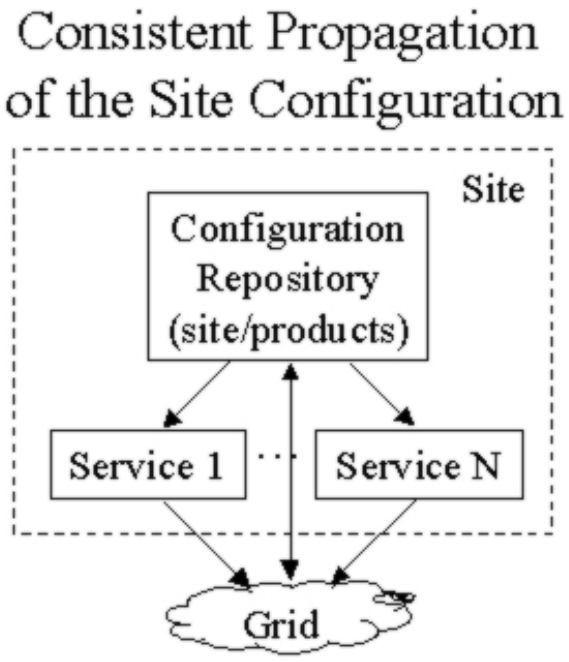


Figure 2.1: Services instantiated on different machines have access to a logically unique site information repository. The services propagate a consistent view of the site to the grid.

1. To gather the configuration, the SAM-Grid has given preference to the knowledge that the administrators have of the system. In other words, the configuration of the SAM-Grid services and products, as well as their hierarchy and relationship with the physical resources at a site, are mainly gathered through interviews with the system administrators. For this reason, a dedicated software tool to drive the interviews is of central importance in the configuration process. A dedicated tool guarantees a consistent integration with the configuration framework. On the other hand, in order for this tool to be usable, it should be easily configurable and flexible enough to fit the logic of all the most relevant interviews. A way to achieve this is for each product to come with a template that drives the tool through the interview. Incidentally, it should be noted that this template could be thought of as a meta-configuration, alluding to the fact that it would be the configuration of how to gather a product or service configuration. The meta-configuration language should address specifically the problem of driving interviews, offering the developers characteristics that could be preferable to a generic language, such as Perl, Python or UNIX shell. Such a tool should satisfy the following requirements:

   (a) it should allow the expression of simple logic, such as loops and branches in the questions

   (b) it should provide facilities to determine the best default to a question, considering previous answers as well as the configuration of other services and resources.

   (c) the structure of the template should reflect the final configuration of the given product, a characteristic useful to ease maintenance and generally difficult to achieve with generic scripting languages.

   Figure 2.2 shows a diagram of the configuration process. Section 2.1.3 describes in detail a tool that uses a meta-configuration language to drive the configuration interview.

2. The SAM-Grid organizes the configuration of its services and resources,

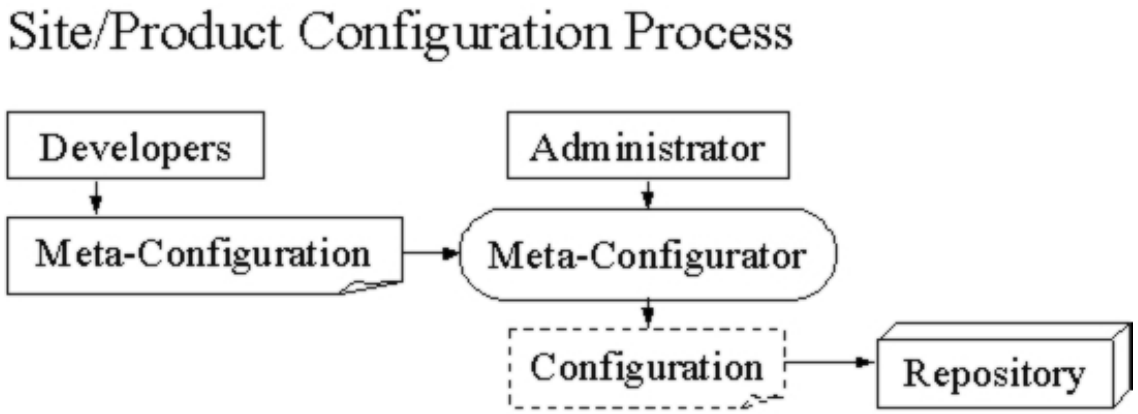


Figure 2.2: Developers provide meta-configuration templates with every grid product. These templates are used as input to the meta-configurator, which drives the configuration interview with the site administrator. The output of the interview is the configuration document, which is stored to a logically unique site repository. Both the meta-configuration template and the configuration are represented in XML. This mechanisms is used for product and site configuration.

as well as their mutual relationships, breaking it down in sub-domains, corresponding approximately to the participating sites and institutions. Although other breakdown structures are, in principle, also possible, ranging from a configuration model where each service and resource is independent to a completely central model, the site-centric approach follows naturally considering the importance of site autonomy. Hence, in this context, the boundaries of a site correspond to the ownership boundaries of the resources. On the other hand, defining all the configuration parameters of a site in a single document becomes verbose and soon difficult to manage. Therefore, a modular approach is preferable, since many parameters are only meaningful to the service that uses them. In other words, it is beneficial to organize the information in a single site description that outlines the relationships between services and resources, while the internal details of their configuration are deferred to different units. As it is possible to organize the configuration of the whole grid in different sub-domains, similarly there are multiple ways of defining such units. Possible models range from grouping the information relative to a service in a single document, to producing a configuration structure for every tool and program that composes the service. Then again, software is typically organized in products, which group together tools and programs that cooperate to accomplish a well defined set of tasks. Thus, the configuration of the SAM-Grid services is organized according to their natural breakdown in software products. Examples of SAM-Grid products are jim_client (job management client, sec. 3.2), jim_broker_client (job management submission service, sec. 3.3), jim_broker (job management resource selector, sec. 3.4), xmldb_server (the Xindice XML database sec. 2.1.2), etc. The organization of products and site configuration are described in sections 2.1.5 and 2.1.6 respectively.

In order to promote the maintainability of the system, both site and product configuration should be manageable using a single set of tools. This toolkit should provide an interface to the configuration framework,

which other programs, such as the meta-configurator, could use. Moreover, although it should be able to access remote configuration repositories, it should also provide robustness in case of occasional network access problems, in order for the SAM-Grid not to trade accessibility for reliability. In other words, it should provide some form of local data caching. On the other hand, such a system faces the problem of maintaining the synchrony between local and remote data.

Section 2.1.4 describes the tool that manages the configuration of the SAM-Grid. This tool is used to manage both the product configuration (sec. 2.1.5) and the site configuration (sec. 2.1.6).

### 2.1.2 Basic Configuration System Architecture

The SAM-Grid configuration infrastructure represents configurations in XML format, which addresses the requirements outlined at the beginning of section 2.1.1 as follows. It addresses the concerns relative to the large parameter space, since XML naturally expresses context-based structures and, therefore, easily allows the definition of name spaces for the parameters. At the same time, available to the community are a set of querying and transformation languages, such as XPath, XUpdate, XSLT and XQuery, which address the problem of efficiently accessing and manipulating XML documents. The challenge of allowing remote and concurrent access to the configuration is addressed by storing the XML configuration in XML databases. All of the XML databases available today support remote data management via transport mechanisms such as CORBA, XMLRPC and SOAP. Furthermore, they provide database semantics for concurrent client accesses via the use of emerging standards such as XUpdate. For the SAM-Grid deployment, the database that we have chosen is Xindice [81], a software developed within the XML project of the Apache Software Foundation. Xindice is implemented as a java servlet, that we generally run using the Tomcat Servlet Engine.

In the SAM-Grid deployment scheme, every site makes available at least one XML database. The SAM-Grid products and services installed locally are configured with the URL of the site database from which they can read

their own configuration as well as the portion of the site configuration that is relevant to them. Thus, the remote accessibility of the configuration repository allows the distribution of the services within the site. In addition, through the use of the SAM-Grid advertisement service (sec. 3.4.2), the URL of each database is registered with a central information collector, hence allowing the traversal of all the grid configuration repositories from anywhere within the grid (sec. 2.2.3).

We conclude this section by discussing the motivations that led us to choose Xindice for the implementation of the SAM-Grid XML databases. Xindice was chosen at the beginning of 2003 as the result of a comparison with other XML database technologies. Xindice provided flexible access by a command line interface and by a remote API via protocols such as CORBA and XML-RPC. This was important because it potentially allowed natural access from both the SAM data handling system and the SAM-Grid monitoring web site. The former, in fact, is built around CORBA and the latter is developed in PHP, which supports XML-RPC. Xindice also provided a client Python API to access the XML-RPC remote interface. This feature was of particular interest as Python is the main scripting language for the SAM-Grid software. Xindice is open source and at the time had an active development community, which had an extensive plan of feature enhancements, most notably support for the XQuery querying language. It already implemented most of the specifications from standards such as XPath, for document querying, and XUpdate, for document updating. Today we can say that the product meets our expectations in terms of performance and ease of access. On the other hand, in the past two years it was not actively developed and there is a lack of stable releases. In addition some limitations of the software were not clearly documented and this required the redesign of some features. For example, Xindice supports the concept of "collection" of documents. A collection is a database container optimized to provide efficient access to information from multiple document with a single XPath query. In order to provide efficient management of redundant data for nested collections, feature that minimizes the probability of database corruptions, Xindice opens a file descriptor for every collection in the database. On the

other hand, on Unix this implementation limits the total number of collections to the total number of file descriptors, typically 1024 for Linux. We hit this limit using the database for the push model monitoring (sec. 2.2.3), as we initially created a collection for every grid job that we ran at a site.

### 2.1.3 The Configuration Process

In this section, we describe mechanisms adopted by the SAM-Grid to gather the configuration of products and of resources and services at a site. As discussed, the SAM-Grid favors gathering the configuration of the system through interviews with the administrators. Developers provide template interviews in a meta-configuration language, discussed in detail hereby.

While the literature on system configuration languages is abundant [82, 83, 84], the study of meta-configuration languages is far from being fully explored. The SAM-Grid configuration framework uses a prototype tool developed by the team, called the meta-configurator, which is a first attempt to address the requirements outlined in section 2.1. The template consists of an XML document, where the tags and attributes define the name of the configuration parameters. The logic of the interview is defined at each tag with special attributes, which are interpreted as directives to the meta-configurator.

Figure 2.3 shows an example from a real template, shortened from the real version for the sake of clarity. As the outermost "site_configuration" tag hints, this is the meta-configuration used to gather the information relative to the services and resources at a site. The nesting of the tags represents the hierarchy of the corresponding parameters in the final configuration, while the attributes that begin with "cardinality" define the minimum and maximum number allowed of each parameter. In the example above, a site configuration is composed of a unique schema version and site name, and at least one cluster, characterized by a name and architecture. In turn, each cluster can have zero or more gatekeepers, characterized by a certain location, and zero or more SAM stations, each with a name and belonging to a certain experiment. The value of the "element-description" attribute

```
<?xml version="1.0"?>
<site_configuration cardinalityMin="1" cardinalityMax="1">
  <schema cardinalityMin="1" cardinalityMax="1" version="1_0"/>

  <site cardinalityMin="1" cardinalityMax="1"
    element-description="Enter the name of the site"
    name="inquire-default,FNAL"/>

  <cluster cardinalityMin="1"
    element-description="Enter the name and architecture of each cluster."
    name="inquire-default,SamGrid"
    architecture="inquire-default,Linux+2.4">

    <gatekeeper cardinalityMin="0"
       element-description="Does this cluster run a gatekeeper?"
       location="inquire-default,cat,hostname,:2119"/>

    <station cardinalityMin="0"
       element-description="Enter the station name and experiment."
       name="inquire-default,fnal-farm"
       experiment="inquire-default,d0"/>

  </cluster>

</site_configuration>
```

Figure 2.3: The meta-configuration template used to configure a SAM-Grid site (simplified)

is the question asked when "inquiring" the value of the particular parameter. Default values are determined by analyzing the comma-separated list of operators and values right after the "inquire-default" keyword, as a stack in inverse polish notation. Typically used operators include "hostname", to determine the fully qualified host name of the machine, "cat", to concatenate two strings, "exec", to run generic UNIX shell commands, "set" and "get" to manage internal variables. When the minimum cardinality of a tag is zero, the meta-configurator asks whether the user wants to define the corresponding parameter or not, thus creating a branch in the interview. In the affirmative case, nested tags are inquired about recursively. The same question is then repeated as many times as indicated by the maximum cardinality of the tag, thus creating a loop in the interview.

Figure 2.4 shows an interview derived from the template of figure 2.3. The configuration resulting from this particular interview is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<site_configuration>
  <schema version="1_0"/>
  <site name="FNAL"/>
  <cluster architecture="Linux+2.4" name="SamGrid">
    <gatekeeper location="samadams.fnal.gov:2119"/>
    <station experiment="d0" name="fnal-farm"/>
    <station experiment="cdf" name="samgrid-test"/>
  </cluster>
</site_configuration>
```

Comparing this configuration and its relative meta-configuration (fig. 2.3), their structure looks similar in terms of tag names and hierarchy. Moreover, we have found that we could use this tool for all the interviews of the SAM-Grid services, generally implementing reasonably short and readable templates.

We believe that this prototype is a step in the right direction, although the tool could be improved by implementing at least two additional features.

1. **New operators**: it would be interesting to extend the number and the

```
---------------------- site_configuration  ----------------------
---------------------- schema  ----------------------
What is the value of 'version' of 'schema' ?
The value of 'version' is set to '1_0'

---------------------- site  ----------------------
Enter the name of the site
What is the value of 'name' of 'site' ? [FNAL]:
The value of 'name' is set to 'FNAL'

---------------------- cluster  ----------------------
Enter the name and architecture of each cluster.
What is the value of 'architecture' of 'cluster' ? [Linux+2.4]:
The value of 'architecture' is set to 'Linux+2.4'
What is the value of 'name' of 'cluster' ? [SamGrid]:
The value of 'name' is set to 'SamGrid'

---------------------- gatekeeper  ----------------------
Does this cluster run a gatekeeper
Do you want to configure gatekeeper ? : [no]yes
What is the value of 'location' of 'gatekeeper' ? [samadams.fnal.gov:2119]:
The value of 'location' is set to 'samadams.fnal.gov:2119'

---------------------- gatekeeper  ----------------------
Does this cluster run a gatekeeper
Do you want to configure another gatekeeper ? : [no]

---------------------- station  ----------------------
Enter the station name and experiment.
Do you want to configure station ? : [no]yes
What is the value of 'name' of 'station' ? [fnal-farm]:
The value of 'name' is set to 'fnal-farm'
What is the value of 'experiment' of 'station' ? [d0]:
The value of 'experiment' is set to 'd0'

---------------------- station  ----------------------
Enter the station name and experiment.
Do you want to configure another station ? : [no]yes
What is the value of 'name' of 'station' ? [fnal-farm]: samgrid-test
The value of 'name' is set to 'samgrid-test'
What is the value of 'experiment' of 'station' ? [d0]: cdf
The value of 'experiment' is set to 'cdf'

---------------------- station  ----------------------
Enter the station name and experiment.
Do you want to configure another station ? : [no]

---------------------- cluster  ----------------------
Enter the name and architecture of each cluster.
Do you want to configure another cluster ? : [no]
```

Figure 2.4: An interview derived from the template of figure 2.3.

behavior of the operators available to determine the default answer to a certain question. The following are examples of possible extensions.

- *Context-sensitive commands*: the exec operator allows the execution of shell commands, but passing the current context to the shell can be done only with the complicated and error prone process of building the command by concatenating strings, using the meta-configurator operators themselves. A typical use of this feature is executing a certain script with arguments that depend on the answers given to previous questions.
- *"If-Then-Else" construct*: today this construct is generally implemented using the "exec" operator and the underlying shell "if" statement, but, as noted in the example above, it is difficult to make it work with the meta-configurator context.
- *Easy access to the configuration of other products*: today this also can be done with the "exec" operator. As in the examples above, though, this mechanism is convoluted.

In summary, we believe that even if the flexibility of the "exec" operator makes it possible to deal with most of the cases of interest, it would be beneficial to implement more specialized operators to address, at a minimum, the three issues stated above.

2. **"If-Then-Else" semantics**: it would be interesting to extend the meta-configurator to include "If-Then-Else" semantics in the creation of the configuration. As discussed, the prototype allows loops, to repeat a nested set of questions, and simple branches, asking the user whether to inquire about a nested set of questions. In some cases, it would be useful to ask different questions depending on previews answers. For example, in the interview described above, without restructuring the schema, it is not possible to ask different questions depending on whether the SAM station is part of a CDF or a DZero installation.

### 2.1.4 The Configuration Management Tools

In section 2.1.1, we have investigated how to organize configuration information for grid services and resources. We discussed how the SAM-Grid organizes the configuration in two broad categories. The "product" configuration gathers parameters that ultimately characterize the services of the SAM-Grid. The "site" configuration describes the relationship between services and resources at a site. To promote ease of maintenance of the system, both categories of configuration are managed via a single tool, described hereby.

The SAM-Grid configuration tool is packaged in a software product (jim_config) that provides a command line interface and a python API to manage configurations. The tool implements two primitives for the interaction with the configuration repository: "store" and "get".

A typical "store" interface accepts as input the configuration document as a file in XML format. The consistency of the XML document is automatically checked by the XML database. The tool stores the document in the XML database as well as on the file system. This second copy was initially introduced for fault tolerance: in case of failures of the Xindice database, we wanted to provide minimal functionalities of the grid services. The ability of reading configurations from the file system, though, turned out to be convenient when establishing the job environment (4.3). The environment preparation service provided scalable file transfer mechanisms, which we could use to propagate configuration information. This optional transport mechanism was an alternative to accessing the XML database and allowed the control of the load of the database. The current implementation of the configuration tool does not provide mechanisms to check the synchrony of the database and file system copies. When the database is up, its copy is used when a retrieve request is made. The design of the next version of the tool includes an automatic synchronization mechanism based on the time of last update of the documents. In practice, in the past two years, we've been affected by synchronization problems only about three times, which justifies the low priority in the implementation of the synchronization mechanism.

The "get" primitive is implemented by two interfaces. The first allows the retrieval of a complete configuration document. The second accepts as input an XPath statement and provides access to single attributes within the document. This latter interface is the one that is mostly used within the SAM-Grid software, as it allows direct access to single configuration parameters.

Despite the benefits of treating site and product configurations symmetrically, their differing usage by the rest of the system poses different constraints on each of them and their management infrastructure. The first difference is the organization of the configuration parameters. For the site configuration, it is crucial to have unambiguous parameter names, organized in a well thought out schema that directly represents the grid view of the relationship among services and resources. Clarity of the organization of these parameters is of fundamental importance because many different services, developed by different people, need to access them. On the other hand, for the configuration of a software product, such care is not always necessary, as generally its parameters are used almost exclusively by the product itself. The second difference between product and site configurations is that products are typically maintained on a system via a product management system, while a site configuration is not. This means that a generic tool that manages product configurations needs to be able to interface to different product management systems, a requirement that simply does not apply to a site configurator. These differences are discussed in more detail in the next two sections.

### 2.1.5 Product configuration

The role of a configuration system is the management of the parameters that influence the behavior of the product. On the other hand, products are generally maintained on a machine via a product management system. The responsibility of a product management system is assisting with the process of installation, upgrade, deletion, etc., and maintaining the metadata associated with each product, such as version, building platform, etc.

The information relative to the product configuration should be organized according to the relevant metadata associated with the product by the product management system. This does not mean that the configuration infrastructure should duplicate these metadata in the configuration of the product. Rather, it should interface to it, being able to deal with concepts like product version, installation machine, building platform, and so on. In fact, information relevant to the product management system may not be relevant to the product configuration. For example, the physical location on the file system of the programs, or the name of the installer, or yet the date of installation, should not affect the software behavior and, therefore, the values of its configuration parameters. On the other hand, this metadata is in general included in almost every product management database.

Another important aspect is the management of the configuration during a product upgrade or downgrade. The installer, in fact, should not be required to go through the configuration procedure, if it is reasonable to assume that the values of the configuration parameters did not change. Of course, there are cases where this assumption does not hold, for example, when the configuration parameters or their semantics changed due to some software development in the product.

The SAM-Grid configuration manager organizes the product configurations according to the product name, version, a generic qualifier, and the machine where the service runs. This last parameter, in fact, is especially relevant for the sites that run multiple instances of the same service and use a single XML database. Software upgrades and downgrades are managed by introducing a versioning system in the configuration template, which is the document that defines the product configuration schema (sec. 2.1.3). Subsequent product releases with compatible configurations are given the same template version, and, in case of upgrade or downgrade, the system clones the previous configuration to define the new one. Conversely, if the template versions are different, the installer is prompted to reconfigure the product. The current product configuration manager is interfaced only with one product management system, the Unix Product Support (UPS) [85].

This system is widely available at Fermilab and collaborating institution and today there is not a high incentive in the integration with other systems, such as RPM or Pacman [86], despite our interest in testing the flexibility of the configuration framework.

### 2.1.6 Site configuration

The site configuration describes services and resources, as well their relationship, within a certain ownership boundary. This configuration is used by other services to represent and interact with the sites and affects the design of grid services, such as monitoring or brokering, as well as other basic infrastructures, such as the job description language. As of today, the grid community has not yet defined a standard for the description of the site configuration. One of the most interesting ongoing studies of the subject is represented by the GLUE (Grid Laboratory Uniform Environment) schema [87], which will be adopted by the Large Hadron Collider (CERN, Geneva, Switzerland) Computing Grid Project (LCG) [37], the Open Science Grid [88], and others. The GLUE schema describes the entities at a site at a very fine-grain level, using UML notation to represent the relationship among them. For each category, the entities considered not only include high-level grid services and standard resources, such as gateway nodes, storage elements or computing clusters, but also traditionally lower-level resources and their characteristics, such as single nodes in a cluster, their memory, local disk size, processor speed, etc. Figure 2.5 shows a simplified diagram of the GLUE schema version 1.1 for the computing services. This diagram is described later in this section.

While the studies related to the GLUE schema are of extreme practical interest, we argue in this research that its level of detail may not be necessary to describe a grid site. We believe, in fact, that the most interesting information for users and other grid services is of aggregate nature. Of course, there is a minimal level of detail that the grid infrastructure must be able to provide, especially when dealing with characteristics of principal importance to the user, such as job status or cluster utilization. Despite

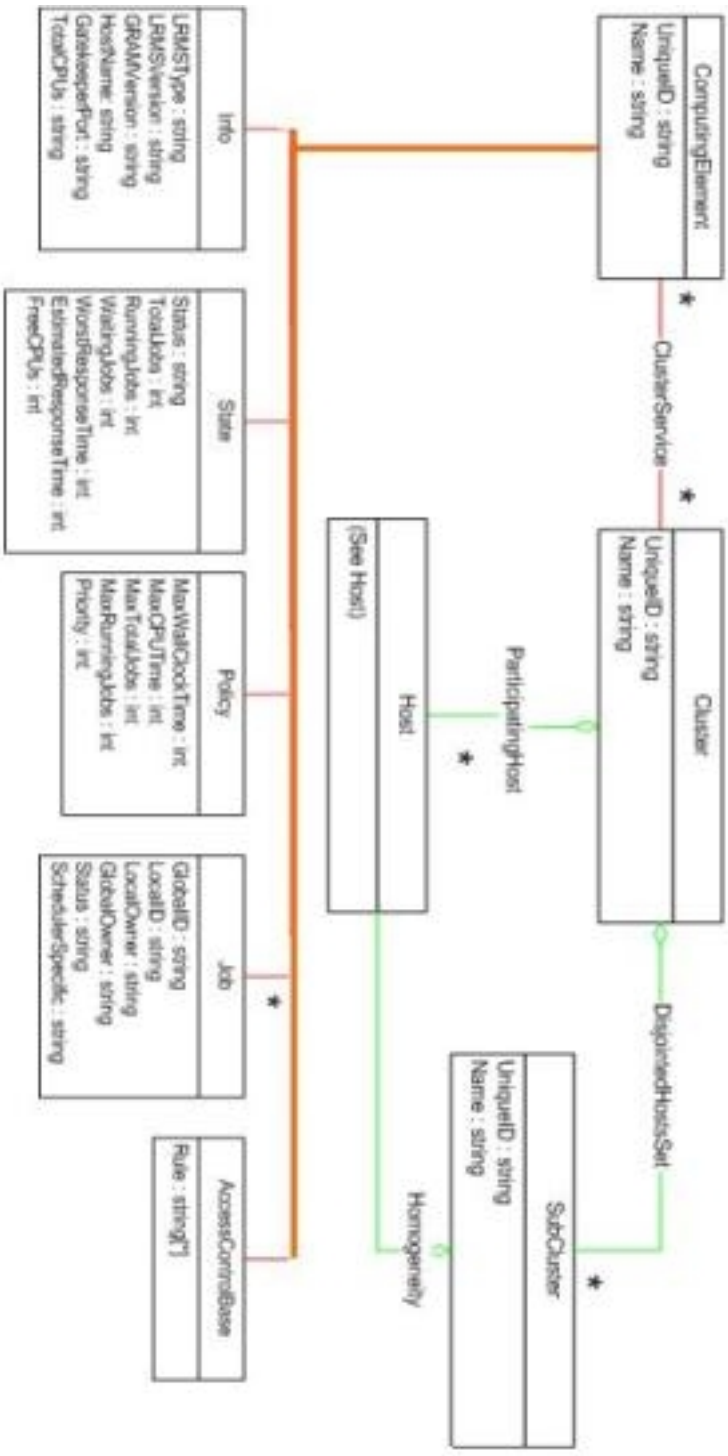


Figure 2.5: The UML diagram of the GLUE Schema v1.1 for the "Computing Element" (partial diagram)

the arguable usefulness of these local details, we believe that there is little incentive today for the system administrators to keep this information up to date. In fact, maintenance and trouble shooting are generally operated using site-specific tools, which today give a variety of diverse views of the services and resources, views not necessarily compatible with the details sought by these grid schemas. Also, automating the collection of this type of information implies the installation of servers at the worker nodes of a cluster, a practice that in general is contrary to the paradigm of distributed ownership of the resources. Then again, studies of the type of the GLUE schema may lead to the definition of standard quantities of interest, which eventually could be gathered with site-specific implementation adhering to the standards agreed upon by the community.

The site description of the SAM-Grid is aggregate in nature and the entities considered are organized in a simple hierarchical structure, with the following relationships:

- a site can have multiple computing clusters
- a cluster can have multiple grid gateways, implemented via Globus gatekeepers (sec. 1.1)
- a gateway can have multiple grid interfaces to local resource managers, implemented via the Globus job-managers
- a "job-manager" can interact with multiple SAM stations (sec. 1.3)
- a cluster can also provide multiple SAM stations not accessible via the grid i.e. data handling services dedicated to users with the privilege of using directly the local site resources
- a cluster, in addition, can provide local data storages, optionally accessible via SAM

Figure 2.6 provide an example of a real site configuration.

Particularly relevant services that access the site description are the Advertising Framework (sec. 3.4.2), the Monitoring Infrastructure (sec. 2.2.2)

```
<?xml version="1.0" ?>
  <site_configuration>
   <schema version="1_1" />
   <site name="Wisconsin" />
   <cluster architecture="Linux+2.4" name="D0PPDG-Cluster">
     <gatekeeper location="apex.cs.wisc.edu:2119">
       <jobmanager name="jobmanager-samgrid">
         <station name="d0ppdg" experiment="d0" universe="prd" />
       </jobmanager>
       <jobmanager name="jobmanager-runjob">
         <station name="d0ppdg" experiment="d0" universe="prd" />
       </jobmanager>
     </gatekeeper>
   </cluster>
  <local_storage path="/sam/disk" node="apex.cs.wisc.edu" />
  </site_configuration>
```

Figure 2.6: The XML representation of a SAM-Grid site, with a cluster, a gatekeeper and two different job-managers interfacing to the same SAM station.

and the Local Sandbox Management Service (sec. 4.3). The reader is referenced to the relative chapters for more details.

The GLUE schema and the SAM-Grid are two different ways of describing resources at a site. The GLUE schema is organized into two main sections for Computational and Storage services. The diagram corresponding to the Computational services is the most interesting to compare with the SAM-Grid. In fact, in the SAM-Grid, storage services are handled by the SAM data handling system in a uniform and aggregated way. The SAM clients generally require little knowledge of the storage systems themselves, as the data handling services are accessed through the SAM interfaces. On the other hand, the GLUE schema for the "Storage Element" [89] focuses on the characteristics of the storage systems on the grid, with no emphasis on higher-level data handling services.

The Computational Services are described as follows (fig. 2.5). The *Computing Element* is the entry point to a batch queuing system, which is characterized/composed by a set of policies, a certain status (e.g. number of jobs currently running), certain access rights, and so on. The Computing Element provides access to one or more computing *Clusters*, which, in turn, can be served by one or more Computing Elements. A cluster is the aggregation of multiple *Hosts*, which can be organized in homogeneous disjoint partitions, or *Sub-Clusters*, within the cluster. In turn, a host can be described by many different characteristics, which include detailed information such as the architecture, the operating system, hardware characteristics and so on. This part of the diagram is not presented in fig. 2.5 and can be found in [89].

It should be noted that both the SAM-Grid and the GLUE schemas are hierarchical in nature. They both separate services from physical resources, using the *Cluster* as the main element of the physical system. To this regard, the SAM-Grid presents a more aggregated view with respect to GLUE, as it does not attempt to describe the elements composing the cluster. On the other hand, one of the main differences is that the GLUE schema emphasizes the concept of batch queue as the main way to access the cluster. Via a queue, the cluster offers different services to the jobs, such as

queuing and batch policies. The SAM-Grid, instead, substitutes the concept of queue with the concept of "smart" Grid-Fabric interfaces (chap. 4), or *job-managers*. For the SAM-Grid, a job-manager represents a strategy of coordination of fabric services to best serve classes of applications at the site. To the core of this decision lies the fact that the typical applications that run on the SAM-Grid are generally not grid-aware. Therefore, the "smart" interfaces need to interact with the grid on behalf of the applications. This is not a problem for the LCG or the Open Science Grid, and, therefore, for the GLUE schema, as they provide service by design to more mature applications.

## 2.2 The Monitoring and Logging Infrastructures

The monitoring, logging and configuration infrastructures are the three aspects of the SAM-Grid information management component. The monitoring infrastructure captures mostly dynamical information relative to main entities of the grid. Such infrastructure addresses the problem that was introduced in section 1.2.3 of how to monitor jobs, services, and resource usage (sections 2.2.2 and 2.2.3). On the other hand, the historical information relative to the same entities is recorded by the logging infrastructure (sec. 2.2.4). For the domain of high energy physics, the most relevant use of such infrastructure consists in the reproducibility of the scientific results. Data analyses in high energy physics sift through Petabytes of data in order to identify rare particles described by few Kilobytes. The infrastructure must assist the scientists in locating this tiny piece of information after it has been found. This concept was also introduced in section 1.2.3 as the problem of bookkeeping of the data processing activity. In the SAM-Grid system, such bookkeeping is responsibility of the SAM data handling system and it is discussed in section 1.3.

### 2.2.1 Basic Monitoring System Architecture

Most of the interactions with a grid system require knowledge of the system status. Users need to know the status of their jobs and of the resources and services that they (hope to) utilize. Administrators need to check that the services and resources they are responsible for are stable and secure. Developers need to check that the service that they have just implemented behaves according to specifications. The list is long, but in general, any stakeholders rely on their ability to *monitor* the system in order to use it.

The monitoring system becomes therefore the heart of the system. But even assuming that the information to monitor is somehow available, how should this information be provided or accessed? In other words, should relevant entities on the grid expose monitoring interfaces, available to be queried by the consumer of the information? Or should the same entities send their status to the consumers proactively? In other words, what should be the streaming model of the monitoring information?

Two approaches exist. In the first approach, events relevant to the entities of the grid are published to the information repositories, hence adopting a *push* streaming model. This approach is best suitable to record a change of state of these entities, independently from the external interest on the information at the time. The information gathered with this model is generally maintained persistently, as it is always significant with respect to some entity, and it is often used by the logging infrastructure as well. A typical example of information monitored with a push model is the change of status of running applications. Applications can be instrumented to report to the system relevant events, such as when input files are staged, when they are processed, when the output is stored, and so on.

The second approach consists in gathering information upon request, adopting a *pull* streaming model. In this model, the monitoring services maintain the information in transient caches, where the knowledge is built incrementally when needed. This information is generally not maintained persistently, since its gathering is prompted by the interest in the status of the entity at the time and not by some relevant event occurring within the

entity itself. Examples of information monitored with a pull model is the status of running servers, such as the SAM data handling servers. These servers generally expose multiple interfaces, which can be located using already available naming infrastructures. In this context, implementing an additional querying interface and make it available to the pull-model monitoring system is the natural thing to do.

Some requirements apply to both push and pull models. The information repositories must be distributed, as they hold information from entities (services, application, etc.) that are located at different sites. Distributing the repositories promotes the site autonomy. On the other hand, it should also easy to compose information coming from different repositories, since the same consumer is often interested in information held at more than one site/repository.

The SAM-Grid implements both pull and push models for its monitoring infrastructure. For historical reasons, though, they are not built as a uniform system. At the beginning, the monitoring system of the SAM-Grid was designed favoring the pull streaming model. The SAM data handling servers were the most mature part of the system and its core, as most of the high energy physics applications are data intensive. These servers implemented already status interfaces. Also, at the time, pull-based systems, such as the Meta Directory Service (MDS) [92, 93, 94, 95, 96, 97, 98], were very popular in our community. MDS in particular was readily available, as it was distributed with the Globus Toolkit. Using MDS to monitor SAM was the first goal for the SAM-Grid monitoring system (sec. 2.2.2). As the SAM-Grid became more used, though, the need to monitor applications became more pressing. Instrumenting applications to report on relevant events became a user requirement. For an application running on the worker node of a cluster, it is much easier reporting proactively internal events rather than opening and listening to querying interfaces. Hence, we designed and implemented a push-based monitoring architecture based on a network of XML databases (sec. 2.2.3).

It should be noted that other information gathering models are also possible, for example most notably the “heart beat”. In this model, information

is gathered periodically and recorded persistently. The drawbacks of this model are that, for logging, it does not necessarily record relevant statuses of the system, and, for monitoring, it shows by design stale information. On the other hand, this model is ideal to take a series of measurements for quantities that vary slowly with respect to the sampling rate. It is also easy to implement and the SAM-Grid uses it to monitor the stability of the data handling services, using a tool called SAM TV [90].

The community is looking with growing interest to a simple but powerful architecture for monitoring distributed systems. The Grid Monitoring Architecture (GMA) [91] of the GGF consists of three components: *consumers*, *producers*, and a directory service, or *registry*. Producers of information register the information that they provide with the registry. Consumers of information query the registry and locate the producers they seek. Consumers then contact directly the producers in order to gather the relevant data. It should be noted that the GMA can be applied irrespectively of the information streaming model. Sections 2.2.2 and 2.2.3 describe how the pull and push based infrastructures of the SAM-Grid adhere to the GMA or to some variation of the same idea.

Various other technologies have been developed to address the problem of monitoring on the grid. The European Data Grid and, now, LCG use R-GMA [75, 99], which we describe in more detail in section 2.2.3. The Condor team has developed a product called Hawkeye [100]. Zhang, Freschk and Schopf compare these two technologies and MDS for scalability and performance [101]. Another popular monitoring system in the grid community is MonALISA [102], based on Java and Jini [103] technologies. Plale, Dinda, and Laszewski compare the performance of hierarchical versus flat table organization of the information [104]. Plale also compares the performance of a MySQL relational database [105] versus the Xindice XML Database [81] for a set of standard database operations [106]. Incidentally, Xindice is the technology chosen by the SAM-Grid for its XML Databases (sec. 2.1.2).

### 2.2.2 Pull-model Monitoring

The SAM-Grid was built around the SAM data handling system. The SAM system consists logically of two parts: first, a set servers collectively called the *station*, which are installed at the site and manage local data handling resources, such as disks, mass storage systems, the network etc. Second a central large database, which implements the replica and metadata catalogues, records the processing history and the user-defined datasets, etc. (sec. 1.3). When we started designing the Job and Information Management components, DZero had already deployed a couple of dozen SAM stations at collaborating institutions around the world. In order to coordinate the operations and maintenance of such a system, DZero had organized a team of *shifters* based in different time zones, so that user inquiries could be dealt with almost $24 \times 7$. It was important for SAM to provide a scalable monitoring solution that could allow retrieving the status of the SAM servers and resources from any locations. Since the SAM station already exposed internal status interfaces, we designed a system that could pull such statuses upon request [107]. This system was based upon the Globus Meta Directory Service (MDS) [108]. This was a popular implementation at the time and readily available to us, as the Globus Toolkit was part of the middleware that the SAM-Grid deployed. In actuality at later times, the community started criticizing MDS for lack of scalability and flexibility in the description of the data monitored. Neither problem affects our monitoring infrastructure, as we discuss in this section after presenting the details of the system.

The core of MDS is a Lightweight Directory Access Protocol (LDAP) server [109, 110]. The information is gathered by information repositories (GRIS) and it is organized in trees, where each node has a unique identity code (figure 2.7). The uniqueness of the nodes facilitates the composition of information trees coming from different repositories. In other words, subtrees stemming from nodes with the same id are composed as subtrees of the same node. The composition of information is also facilitated by a registration service (GIIS). The architecture of MDS adheres to the Grid Monitoring Architecture of GGF (sec. 2.2.1), where the GRIS component

acts as the information *producer* and the GIIS as the *registry*.

The organization in a tree structure is well suited for a monitoring system where knowledge is built incrementally, a typical pattern when the information is explored by a human being. The information for each node is gathered by launching executables called information providers. The information providers are launched when MDS is queried for the information contained in a certain node. The information in the node is then maintained in transient caches in order to optimize access time and server load [101].

Despite the clean design, the implementation of MDS is nowadays generally considered low quality. In particular, when MDS is organized in a multi-layered hierarchical configuration, people lament problems in the availability of the registration server [111]: typical queries cause the server to hang, while the time out mechanisms do not work properly. MDS has also problems scaling to more than three servers for queries as frequent as 3 Hz. These scalability problems do not affect our infrastructure because we adopt a shallow hierarchical configuration, as we explain later in this section. In addition, in our case, the information that we monitor with MDS typically varies significantly on the time scale of the minute, far slower than the 3 Hz limit reported in [111].

Besides the quality of the implementation of MDS, the LDAP model has been the target of various criticism when used to monitor large distributed systems. First, the data model is considered too inflexible to satisfy the requirements of a large and dynamical environment such as the grid. In fact, the organization of the information tree, or *schema*, is configured statically before starting the server. In reality, we have found that this problem can be in part mitigated by implementing information providers that return dynamic subtrees of information. This technique is described in detail later in this section. Second, LDAP is optimized to retrieve the information of the schema rapidly, but complex queries can be expensive to run. The query language, in fact, is procedural in nature and does not allow the retrieval of information based on more than one node of the schema at the time. In terms of relational database algebra, this means that join operations are not supported. In other words, a query like “retrieve the information

on node X, where X is the value of node Y" cannot be submitted to the system. This is why LDAP is considered best suited for those systems where monitoring data should be gathered incrementally. The LDAP approach to the query above, in fact, consists of an initial query to find X, like "retrieve the value of node Y", and a subsequent query like "retrieve the value of node X". Again, this limitation does not affect the SAM-Grid because MDS is used for human-driven monitoring, where knowledge is typically sought incrementally. In addition, this limitation does not exist for the push model monitoring infrastructure (sec. 2.2.2). This infrastructure is based on a network of XML databases that support the XPath query language. Using this technology, a query to an XML document can be based on the content of other documents in the same database.

The SAM-Grid deploys an MDS per site. The information monitored is aggregate in nature and reflects the status of services distributed throughout the site. The way SAM-Grid uses MDS to monitor remote services is not common. In some grid systems [36, 39], in fact, multiple GRIS are deployed at a site and register with a hierarchy of GIIS. Each GRIS is configured to run the same set of information providers. The information providers report to their GRIS measurements that are local to the machine where they run, such as available memory, file system status, etc. As noted above, this multi-layered configuration lead to potential scalability problems. On the other hand, the information providers of the SAM-Grid use remote access protocols, such as CORBA, to gather the status of non-local services. Because of this, the SAM-Grid generally can deploy a single GRIS and a single GIIS at a site, therefore facilitating the deployment and maintenance of the monitoring infrastructure.

This type of information providers is used for the monitoring of the SAM station. The use of MDS for monitoring the SAM station offers a couple of advantages as compared to the direct monitoring of the station servers. First, it allows uniform access to all the servers of the station. That is, MDS presents the status of the station as a whole, as opposed to the status of its individual components. Second, it decreases the frequency of the status inquiries to each server, thanks to its information caching service.

The caching time has been configured to be shorter than the typical time for which any relevant variation occurs in the station.

Some of the status information of a SAM station is dynamic in nature. Typical examples include the list of data transfer requests and their status, the list of storage resources managed by the station, the list of physics groups allowed to use storage resources, etc. Using MDS to monitor these quantities was inconvenient, as the basic MDS configuration consists of a static information tree, or *schema*, programmed in the LDAP Data Interchange Format (LDIF) [112]. Each node of the tree is filled upon request by launching an information provider, which, typically, returns single measurement values. Since the lists we have mentioned are dynamic, they could not be programmed into a static information schema. In order to overcome this lack of flexibility, we had to develop information providers that generate dynamic information trees in the form of LDIF statement.

Figure 2.7 shows a sample output of the SAM-Grid information provider that returns information on the SAM disks at a site. The number of disks and their sizes varies from site to site and it is provided dynamically.

The schema is derived from the SAM-Grid site configuration, by transforming the XML representation of the site resources to LDIF. This translation mechanism provides for a consistent view of the resources and services at the site, irrespectively of the service that publishes the information (sec. 2.1.2).

### 2.2.3 Push-model Monitoring

The monitoring system of the SAM-Grid had to be able to provide two main categories of information:

1. the status of the services and resources on the Grid. This information was monitored for stability almost $24 \times 7$ by a group of DZero shifters, responsible for the smooth operations of the system.

2. the change of status on relevant entities on the grid, typically user jobs.

```
dn: Sam-View-Name=1327,Sam-View-Group-Name=disks,Sam-View-Group-Name=
configuration,Mds-Software-deployment=SamStation-prd-d0karlsruhe,
Sam-View-Group-Name=details,Mds-Software-deployment=Sam,Jim-Cluster=
Gridka,Jim-Site=Gridka,Mds-Vo-name=d0,o=samgrid
objectClass: SamConfigurationDisks
Sam-Disk-Id: 1327
Sam-Disk-Location: d0.fzk.de:/grid/fzk.de/d0/d0-3
Sam-Disk-FreeMB: 5.2
Sam-Disk-SizeMB: 4194304.0
Sam-Disk-Status: Active

dn: Sam-View-Name=1348,Sam-View-Group-Name=disks,Sam-View-Group-Name=
configuration,Mds-Software-deployment=SamStation-prd-d0karlsruhe,
Sam-View-Group-Name=details,Mds-Software-deployment=Sam,Jim-Cluster=
Gridka,Jim-Site=Gridka,Mds-Vo-name=d0,o=samgrid
objectClass: SamConfigurationDisks
Sam-Disk-Id: 1348
Sam-Disk-Location: d0.fzk.de:/grid/fzk.de/d0/d0-1
Sam-Disk-FreeMB: 1479150.1
Sam-Disk-SizeMB: 4194304.0
Sam-Disk-Status: Active

dn: Sam-View-Group-Name=disks,Sam-View-Group-Name=configuration,
Mds-Software-deployment=SamStation-prd-d0karlsruhe,Sam-View-Group-Name=
details,Mds-Software-deployment=Sam,Jim-Cluster=Gridka,Jim-Site=Gridka,
Mds-Vo-name=d0,o=samgrid
objectClass: SamConfigurationDisksViewGroup
Sam-Disk-Total: 2
Sam-Disk-FreeMB-Total: 1479155.3
Sam-Disk-SizeMB-Total: 8388608.0
Sam-Disk-Search-Status-Code: 0
```

Figure 2.7: The output of a SAM-Grid information provider in LDIF format. The monitoring information refers to the site disks managed by SAM. This output generates three monitoring nodes: two are of type "SamConfigurationDisks", which provides the details of a SAM disk, and one is of type "SamConfigurationDisksViewGroup", which gives a summary of all the disks managed by the SAM station. The position of each node in the whole information tree is determined by the "dn" attribute. The part of the "dn" that is common to all three nodes is determined by the MDS schema.

In principle, the two approaches seem equivalent. The approach that the SAM-Grid uses to monitor servers is pull-based i.e. servers provide their internal status upon request. One may argue that the server could be monitored also using a push-based approach i.e. the server sends its status to some repository whenever the status changes; the monitoring system uses this repository to show the server status. In reality, using one approach or the other is determined by various practical considerations. First, the push-based approach generates network traffic. Monitoring may become a resource intensive rather than a non-invasive activity. Second, what determines a change of status? Some quantities are for every practical purpose not discrete, for example the amount of space available on a disk. The granularity of the thresholds to determine the "reasonable" change of a quantity should be weighted against the amount of messages it generates. Third, not all the information has historical value. Pushing changes of status is convenient because the information can be stored, at least temporarily. Sometimes this is simply not a requirement. Considering these points, the SAM-Grid was designed to monitor servers only upon request.

On the other hand, the SAM-Grid uses a push-based model to record the internal statuses of the applications. Again, following the points above, one may argue that inquiring the status of an application when the users feel the need is a valid approach. In our case, though, first, applications go through a very limited and well defined set of statuses (many are, in fact, state machines). Every time the status changes, it is feasible sending a message: typical applications send a couple of messages every hour. Second, we use these messages to determine important actions, such as the recovery strategy in case of application failure. As such, the messages are stored persistently and the whole history of every application is available to consumers of the information. These are typically users or other applications, such as scripts that help with the operations. In summary, using a push-based approach for the SAM-Grid monitoring application is the adequate solution.

The push-based monitoring system was designed to offer a high level of flexibility in the format of the messages. This was important since we wanted to be able to modify the monitoring events as our understanding of

the system and its metrics evolved over time. In particular for monitoring applications, the events had to be as general as possible, since we could not foresee every possible use of the infrastructure. The XML syntax seemed therefore a good candidate, as we discuss later. In addition the system had to be fully distributed to preserve site independence. In other words, we wanted sites to be able to deploy the system and not to depend on the availability of central servers to use it. We describe the details of the push-based monitoring system hereby.

In general, a push-model monitoring infrastructure has two main responsibilities:

- accepting monitoring events coming from the system
- publishing the events to consumers of the information

The SAM-Grid push-model monitoring infrastructure is based on a variation of the Grid Monitoring Architecture (GMA) (sec. 2.2.1). In the GMA, each producer of information is an independent entity that can be queried by any consumer. In other words, the *producers* are also *publishers* of information. The SAM-Grid, instead, distinguishes between these two roles. Information producers store the *produced* information on a repository, which, in turn, is responsible for the *publication*. The SAM-Grid deploys multiple repositories, each aggregating the information from a different set of producers. Another difference is that in the GMA, each producer registers with the registry. In the SAM-Grid, the repositories register themselves and the characteristics of the producers relevant for the look up of stored information.

The system has been implemented as follows. The information repositories are implemented by a network of XML databases. The choice of using XML trees to represent monitoring events stems naturally from the requirement on the flexibility of the event format. Producers store events to the repository by creating or updating XML documents. The details of the organization in documents depends on the type of producers. We describe the use case of job monitoring later in this section. We provide an

API to assist with the storage of monitoring events. The API accepts XUpdate statements to define the content and internal database location of the events.

The publishing interface supports the retrieval of full documents as well as of set of XML nodes. The set of XML nodes can span multiple documents. The querying language that our repositories support today is XPath. Using XPath, it is possible to retrieve a different set of nodes depending on the content of the database itself. In terms of relational logic, considering the logical equivalence between tables and XML documents, this means that the repositories support the join operator. In other words, a consumer of information can express queries such as "retrieve the nodes from any document that satisfy condition X, where X is a function of the value of node Y from a given document".

Incidentally, the network of databases used by the pull-based monitoring is also used by the configuration infrastructure (sec. 2.1.2). This improves our ability to maintain and support the overall system.

The registration service is implemented by the condor information collector (sec. 3.4.2). Information repositories register attributes about themselves, such as remote access point and site location, and about the producers of information, such as cluster or data handling names and their characteristics. Because the XML databases do not implement a native registration mechanism, the repositories rely on the SAM-Grid site advertisement service. Since the condor collector is central, it is a natural entry point to the whole monitoring information in the system. The collector has resulted in a stable and highly available service, and so far we have not experienced problems related to the scalability and reliability of the registry.

In most grid infrastructures, the monitoring system is the core of the resource selection process. For the SAM-Grid, since grid job management is based on the Condor framework, it is easy to base the resource selection on the information contained in the collector, rather than querying the whole monitoring infrastructure. This practice is more of a convenience, rather than a necessity. The condor match making service, in fact, has mechanisms to call external algorithms during the phase of resource/job matching. Such

mechanisms are used by the SAM-Grid to query the status of the data handling services and can be extended to access the whole pull and push based monitoring infrastructure.

An interesting monitoring system that implements the GMA is R-GMA [75, 99]. Developed for LCG, R-GMA is a distributed system that uses the relational model to handle information. Producers make information available to consumers as relations (tables) and use relations to handle the registration process. The consumers see the system as a global database that can be queried using SQL. Internally, the information is partitioned at the granularity of the producers. The registry pulls the system together by holding the description of such partition.

The registry holds an identifier for each type of table that can be produced. Each producer registers two types of information: the identifier of the table for which it generates information and a logical condition that defines to what rows of the table the information refers. Let us use a simplified version of the information relative to an LCG computing element to explain this concept. The table for a computing element has columns such as country, site, operating system, free CPUs, information update time. A producer for a computing element at the Rutherford Appleton Laboratory (RAL) in the UK will register the identifier for the computing element table and the condition "country = UK and site = RAL". Every consumer that queries the system for informations about the computing element at RAL, UK, will be put in direct contact with the RAL producer by the registry. Despite the fact that the registry is programmed with a static schema that defines the identifiers for each table, there are provisions in the system to handle dynamic information.

The R-GMA and SAM-Grid systems are similar as they are based respectively on the GMA and on a variation of the GMA. On the other hand they differ in a few points.

1. The organization of the information is based on two different models: for R-GMA is relational, for the SAM-Gird is hierarchical. Both use declarative languages to query the information, such as SQL, for R-

GMA, and XPath, for SAM-Grid. Plale, Dinda, and Laszewski [104] argues that writing queries for a hierarchical model requires a deeper knowledge of the organization of the information.

2. In R-GMA each producer implements both the producer and publishing interfaces. In SAM-Grid, instead, the two roles are delegated to the producer and the repository components respectively. Because in R-GMA the information is not aggregated, the system needs a component, the *mediator*, responsible for aggregating on behalf of the consumer the information coming from multiple producers. This task needs particular care for join operations. In SAM-Grid, the information is aggregated at the repository and join operations are handled naturally within the same repository. In our experience, the information at a repository is sufficient for most typical queries. For non-typical queries, the responsibility of aggregating information across repositories has been left, so far, to the consumers.

3. R-GMA do not put emphasis on the pull vs push streaming models. In R-GMA, the streaming model is negotiated when a consumer contacts a producer. The consumer may ask the producer to retrieve the information upon request (pull-model) or to publish it when new data is available (push-model). This difference on emphasis probably stems from the fact that the SAM-Grid has historically concentrated more on integration rather than development, while LCG did the contrary. While SAM-Grid had to integrate different solutions to implement its monitoring infrastructure, LCG had the resources to develop an integrated monitoring system.

4. For R-GMA, the presence of repositories has less relevance than for the SAM-Grid. R-GMA implements the persistency of the information via a special producer, the *DataBaseProducer*, which stores to a back end database monitoring events. On the other hand, these repositories do not act as information aggregators, as it happens in the SAM-Grid architecture, instead.

```
<?xml version="1.0"?>
<global_job Id="sam_hep.westgrid.ca_121506_11389"
            owner="864762865" created="Jan 31 2005 20:16:22"
    created_epoch="1107202582.22">
  <cluster_job Id="14705.1107202528" cluster="Westgrid"
               created="Jan 31 2005 20:16:22"
       created_epoch="1107202582.24">
    <local_job Id="955355" />
    <local_job Id="955356" />
    ...
  </cluster_job>
</global_job>
```

Figure 2.8: In the SAM-Grid a grid job is split in multiple cluster jobs, submitted at different execution sites. Cluster jobs are then split in multiple local jobs. The grid/fabric interface stores the monitoring event describing this hierarchy.

In the rest of this section we describe in detail how the information for monitoring jobs is organized [113]. The hierarchical model naturally represents the hierarchy of job-related monitoring events. In the SAM-Grid, in fact, jobs submitted to the grid are decomposed into multiple instances of cluster jobs, each submitted to a different execution site. The cluster jobs are then in turn decomposed into multiple batch jobs, submitted to the local batch scheduler (sec. 4.2). The grid-to-fabric interface generates a monitoring events that encodes this hierarchy, when the job enters the site. This event is stored in the repository as an XML document. Figure 2.8 shows a real-life example of such an event.

At the time of local job submission, the grid/fabric interface produces also another monitoring event, with information relative to the local job. This is stored at the repository as a separate XML document. Further monitoring events related to this job are stored here. This document contains information about the context of the job in terms of the originating global and cluster jobs. In the current schema, the SAM-Grid monitoring API allows the production of four types of events, represented by the “Status”,

"Progress", "Output", and "Usage" tags. The Status tag reports the status of the job in the queue. The representation of this status is independent from the underlying batch system. The translation between batch system specific and abstract statuses occurs in the batch system adaptation layer (sec. 4.2). These messages are accompanied by time stamps, both in human and machine readable format. The Progress tag is used mainly by the applications to report messages relative to their internal status. These messages are typically used in monitoring web pages and are thought for human consumption. Since the SAM-Grid deals mostly with data intensive applications, the storage of output data has a special relevance. For this reason these monitoring events are produced with the dedicated tag Output. The possible statuses of the output storage varies depending on whether the application is storing physical or "virtual" files. Virtual files are logic entities that are recorded with the SAM database for bookkeeping purposes only and correspond to an intermediate phase of the data processing. Finally, the Usage tag reports parameters related to the CPU consumption of the job process. Figure 2.9 shows an example of a local job XML document.

The organization of the monitoring information in separate documents is the result of direct experience. Two different types of organizations, in fact, have been considered in earlier versions of the system and abandoned. Initially, all the information was stored in a single document. This often resulted in large documents (hundreds of KB), which caused the degradation of the querying performance. We also considered the organization of each grid job in a separate collection of documents. Xindice, though, has a limitation on the number of collections supported. This consists in the total number of file descriptors available to the system, which is always small compared to the number of jobs in the system. In addition, Xindice does not support queries that gather data across collections. The current organization is the result of the "lessons learned" from the deployment of the SAM-Grid for real applications.

```
<?xml version="1.0"?>
<local_job Id="955356" created="Jan 31 2005 20:16:25"
           created_epoch="1107202585.57">
   <global_job>sam_hep.westgrid.ca_121506_11389</global_job>
   <cluster_job>14705.1107202528</cluster_job>
   <status time="Jan 31 2005 20:16:25" time_epoch="1107202585.62"
   code="submitted" />
   <status time="Jan 31 2005 20:16:45" time_epoch="1107202605.26"
           code="pending" />
   <progress created="Jan 31 2005 12:18:22">
      Launching user executable on ice30_5
   </progress>
   <progress created="Jan 31 2005 12:20:19">
      Finished staging job files
   </progress>
   <status time="Jan 31 2005 20:21:50" time_epoch="1107202910.63"
           code="active" />
   <progress created="Jan 31 2005 12:30:42">
      Finished staging input file all_0000176852_005.raw
   </progress>
   <progress created="Jan 31 2005 12:33:51">
      Finished setup, launching mc_runjob
   </progress>
   <output_file
    name="/scratch/95...westgrid.ca_121506_11389_WestGrid_NumEv-10"
    time="Jan 31 2005 20:41:21" time_epoch="1107204081.53"
    state="stored" />
   <progress created="Jan 31 2005 12:41:26">
      User executable exited with code 0
   </progress>
   <usage elapsed_time="23:03.33" kernel_cpu="33.22 sec"
          user_cpu="39.39 sec" per_cpu="5%" />
   <status time="Jan 31 2005 20:42:22" time_epoch="1107204142.35"
           code="done" />
</local_job>
```

Figure 2.9: The monitoring information associated with a local job. The system organizes it in a separate XML document.

### 2.2.4 The Logging Infrastructure

The information saved in the SAM-Grid logging infrastructure can be decomposed in three major categories. First, the data processing history, which consists of information of fundamental importance for the reproducibility of the scientific results. This information is recorded in the SAM relational database as a side effect of using the data handling services (sec. 1.3). It includes an identifier of the processing application, as well as the dataset processed with all its relevant metadata. Second, the history of the statuses of main entities of the grid, such as services and jobs. This information can be used to study the system, but do not participate in the accounting of the scientific discoveries. The job status monitoring information (2.2.3) is part of this category as are a series of SAM tools [114, 90]. Third, the debugging messages logged by each service on the grid. This information is of interest to developers only and in case some error condition occurs. The infrastructure consists of a set of distributed logging servers, capable of logging unstructured messages. The information transfer is unreliable but not blocking by design, using the UDP network protocol. This system could be improved by making the logging servers more easily distributed and allowing the handling of structured method [115].

The SAM-Grid did not adopt a uniform solution to address all the categories of information, mainly for historical reasons. On the other hand, the system allows a semi-uniform access to the information for the users by the use of various web interfaces. For the SAM-Grid monitoring system, the web site [116] uses a set of PHP [117] scripts that present a consistent hierarchical extensible view of the whole system. This development is the result of a collaboration with NorduGrid [39], who shared with us their initial implementation of their web monitoring pages.

# Chapter 3

# The Job Management Component

At the end of 2001, the DZero experiment was doing about 80% of its computing at Fermilab and 20% at the collaborating institutions. This imbalance was not uncommon for High Energy Physics experiments and the reasons behind it were mostly historical (chap 1). The growing needs for computing together with this disproportionate repartition of the computing load, caused the Fermilab resources to become over-subscribed.

The problem was addressed following two complementary strategies. The first consisted in decommissioning high-cost and hardly-extensible systems in favor of cheaper and more scalable solutions. At the time, most of the user analysis was run on the "d0mino" super-computer, an SGI Challenge with 176 processors and about a dozen Terabytes of disk attached. The experiment was happy with the performance of this machine, but it was costly in terms of maintenance fees and hardware upgrades. In 2002, DZero built "CAB", a cluster of commodity computers based at Fermilab. This solution was an initial attempt to offload the d0mino super-computing and to promote the extensibility of the infrastructure. In the same years, in fact, the CDF experiment had already adopted a similar computing infrastructure, the Central Analysis Facility (CAF) [118], and their experience was positive.

The second strategy consisted in developing a global computing infras-

tructure with the goal of lowering the dependence of the experiment on the Fermilab facilities. At the time, DZero had already deployed about a dozen SAM stations (sec. 1.3), the agents that, among other things, were capable of distributing the data stored at Fermilab to remote sites. The SAM system was actively maintained and and user support was a very high priority. This system enabled the use of remote computing resources, but did not solve the problem of presenting to the users a uniform computing environment. The user had to know and consider the resource configuration at the sites where s/he worked. For example, managing jobs was done differently when using the Fermilab or the remote facilities. In summary, the users lacked a layer of abstraction on top of the resources that allowed seamless access to them via a common interface.

The Job and Information Management (JIM) project was started in late 2001 to achieve globally distributed computing for DZero and CDF, integrating standard middleware with the SAM data handling system and developing new grid technologies, when needed. The whole suite of components for job, data and information management was called the SAM-Grid. The job management component of JIM is the subject of this chapter.

The following is the list of the requirements on the job management infrastructure.

1. **Site Autonomy**: the infrastructure must foster site autonomy, as to satisfy the grid paradigm of distributed ownership of the resources. This means that the grid must not impose any specific choice of local fabric management systems, such as local schedulers, intra-cluster data distribution mechanisms, or monitoring tools. In particular, the grid should not impose the presence of any software or running processes at the worker nodes of the local cluster. Maintaining this software on hundreds of nodes, in fact, is a burden that few site administrators are willing to carry. First, every new process running on the cluster increases the probability of bug exploits and security breaches. Denial of service attacks launched to high profile targets by a large facility is on of the worst case scenarios of most administrators. Only very

stable, widely distributed, highly tested software generally is considered low-risk enough to be acceptable. Second, some versions of the software may turn out to have side effects on the system or to be intrusive, that is resource intensive. This may lead to instabilities of the system and node crashes. Any new piece of grid software is generally looked at suspiciously for bugs or inefficiencies that may lead to such system instabilities. Third, there would be the extra work of updating regularly the software as new releases come out. In reality, this turns out to be the least of the concerns, as several tools are available to accomplish this task. In addition, this is considered a standard responsibility of site administrators. For these reasons, the grid has grown acceptance in the community maintaining the paradigm of distributed ownership of the resources. In other words, the grid should interface to site resources through a few well-identified machines.

2. **No Continuous Network Connectivity**: the system should not require continuous network connectivity with the machine from which the user manages the job. In other words, the user should not need to have login access to special job management machines, but rather the system should provide some lightweight user interface to a core of highly available job management services. This user interface software, in principle, should be able to run on the user's laptop. Section 3.3 describe the solution implemented by the SAM-Grid.

3. **Reliability**: the job management infrastructure should be reliable, handling the job instance persistently, and guaranteeing the retrieval of any output and/or errors of the application and the middleware. Incidentally, error retrieval is also of fundamental importance during the development of the infrastructure itself. This is in fact the time when errors and log files are studied with particular care in order to discover and fix software defects. Section 3.3 addresses the requirement of reliability on the SAM-Grid.

4. **Automatic Resource Selection**: the system should provide an au-

tomatic resource selection service. This service, sometimes called "broker", should analyze the job requirements and select the best resource available for the job, according to some dynamically configurable algorithm. This service should include the possibility for the user to reference site resource attributes in the description of the job, for example to enhance the job environment. Given the complexity and dynamic nature of a grid the job management system should be able to react to suboptimal decisions of the broker and consider the resource selection process mainly as a "recommendation". Section 3.4 and its relative subsections present the SAM-Grid solution to resource selection.

5. **Fault Tolerance**: The system should implement some form of fault tolerance, especially to temporary disruption of service. During job submission, for example, there should be a mechanism for automatic resubmission to the same resource, with capabilities of asking the broker to select a different one in case of prolonged unavailability. Sections 3.3 and 3.2.4 discuss fault tolerance for the SAM-Grid.

6. **Structured Job Management**: it should be possible to automatically execute jobs that expose to the grid their internal interdependencies. We call this type of jobs "structured", as opposed to "unstructured" jobs, which are treated as atomic processing units. The job management infrastructure should execute the atomic jobs composing a structured job in the right order, possibly relying on the data handling system to handle input and output. It should also provide some mechanism to check the success of each atomic job not only by its exit status, but also in the context of the whole structure. In addition, in case of failure, it should allow the resubmission of the jobs that failed or had not yet run. Section 3.2.4 presents structured job management on the SAM-Grid.

7. **Application Diversity**: The job infrastructure should be able to handle the major high energy physics applications, despite their dif-

ferent requirements on the services and resources of the grid (sec. 3.1). Because of this wide range of application requirements, a general-purpose grid tends to be inefficient in handling resources. Section 3.2 and its relative subsections explore the problem of how much application-specific knowledge the middleware should have to optimize the use of resources.

8. **Performance**: The system should comply with minimum performance requirements for metrics such as the number of jobs that can be submitted per unit time or the "cost" of job submission. In this regard, we want to stress that for CDF and DZero, the former is not a concern in the case of reconstruction and montecarlo jobs (sec. 3.1). In fact, as previously discussed, these jobs run for hours or dozens of hours and are submitted by a small number of experts in a coordinated fashion. Thus, this community is willing to wait minutes for every single job submission, if this means running more consistency checks over the job request, and therefore increasing the probability of terminating the job successfully. Chapter 4 presents major bottlenecks in the performance of the SAM-Grid system and the solutions implemented to overcome them.

The SAM-Grid addresses these requirements with an infrastructure composed of four major components, organized in a three-tier architecture, as shown in figure 3.1.

The first tier consists of a thin layer of software that interfaces the user to the system. For job submission, this user interface should accept two pieces of information: first, the description of the job, specified in a high-level language meaningful to the physicists; second, and optionally, the set of all the files necessary to run the application i.e. executables, libraries, configurations, etc. After submission, the user interface should assign the job a unique identifier, usable as a handle for any further management. Aside from when managing the jobs, the user interface should not mandate network connectivity in order to comply with the above requirements. The envisioned multiplicity of this tier is up to one per user i.e. up to hundreds

of instances. The functionalities and implementation of this software layer is described in detail in section 3.2.

The second tier maintains a persistent queue of grid jobs and, being interfaced with the first tier, acts as a mediator between the user and the job instance. Because this tier acts on behalf of the user, submitting the job to a resource capable of executing it, it is also called the "submission" tier. The envisioned multiplicity of this tier is of a few instances per nation, or, in other words, a few dozens throughout the grid. The submission tier is described in detail in section 3.3.

The third tier consists of the sites that ultimately run the jobs. These sites must provide computing and storage resources, tied together by a set of local and grid services. In jargon, these local resources are sometimes called the grid "fabric". The fabric management and execution site grid services are described in chapter 4. Execution sites advertise their characteristics to a Resource Selector service, including such information as computing cluster availability and their gateway entry point, the reference to the local data handling services, the local monitoring system URL, etc. The Resource Selector acts as the glue of the job-handling infrastructure, recommending to the submission sites what resource best matches the requirements of each job. Today this service is centralized. Nevertheless, should scalability problems arise, the service could be easily decentralized. The resource selector is described in section 3.4.

## 3.1 Typical High Energy Physics Applications

Modern high energy physics experiments, such as DZero and CDF, typically acquire more than one TB of data per day and move even ten times as much. To give an example, during the past year the SAM system has stored 400 TB of data for DZero alone. Aside from the stream of data from the detector, various other computing activities contribute to the one TB of data stored per day. While the data handling system is responsible for the management of the large data volume, the job management infrastructure provides facilities to assist with the execution of the computation activities.

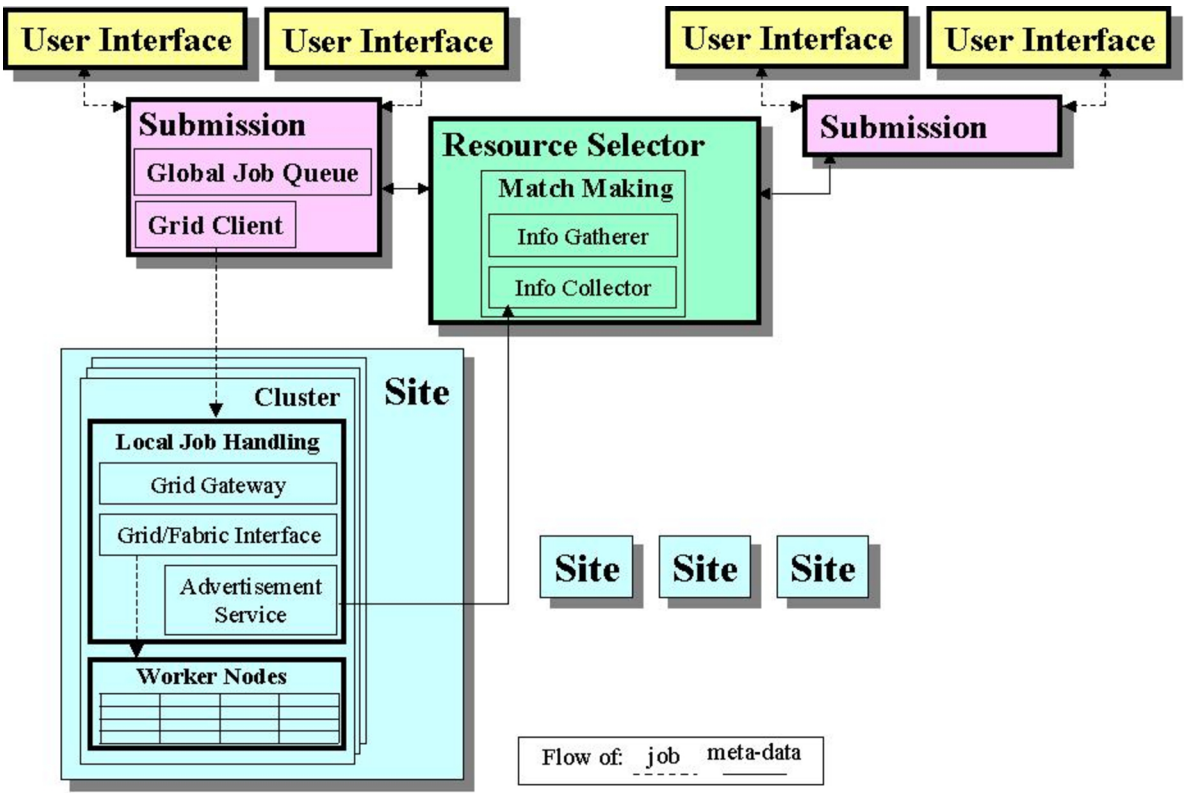


Figure 3.1: The SAM-Grid job submission management infrastructure is based on a three-tier architecture. The first tier is the user interface (top), a thin layer of software used to manage the jobs. The second tier is the submission site, a suite of services that maintain the queue of grid jobs, mediate the interaction between the users and the remote resources and schedule jobs on behalf of the user. The third tier is the execution site (bottom), where the jobs are run on the available resources (the figure shows only local job handling resources and services, for clarity). The resource selector collects the resource characteristics advertised by each execution site, and assists the submission site in deciding where to run each job.

The characteristics of these activities impose a wide variety of requirements on the resources and services that handle the jobs. Understanding such requirements was a fundamental step to implement the JIM infrastructure. This section discusses these requirements for three typical computing activities of high energy physics.

The activities discussed are data filtering, also called data "reconstruction", the production of simulated events, and data analysis. This third activity broadly consists of the selection and the statistical study of particles with certain characteristics, with the goal of achieving physics measurements. It should be noted that the first two activities are indispensable for the third one. During data reconstruction, the binary format of events from the detector is transformed into a format that more easily maps to abstract physics concepts, such as particle tracks, charge, spin, et cetera. The original format, called in jargon "raw", is instead very closely dependent on the hardware layout of the detector, in order to guarantee the performance of the data acquisition system, and is not suitable for data analysis. On the other hand, the production of simulated events, also called "montecarlo" production, is necessary to understand the characteristics of the detector either related to the hardware, such as the particle detection efficiency, or to physics phenomena, such as signal to background discrimination.

These three typical activities, which ultimately correspond to software application families, differ among themselves principally, but not uniquely, by the usage of the computing resources. The communities that run the software range from a handful of almost dedicated experts in the case of reconstruction and montecarlo activities, to potentially the whole physics community in the case of data analysis. The typical duration of a single reconstruction or montecarlo job is dozens of hours, while data analysis ranges from a few minutes to days, depending on the problem studied. All the activities are CPU intensive, but while both reconstruction and analysis are highly I/O intensive, montecarlo is not. In fact, montecarlo almost never requires any input data, while for reconstruction and, especially, analysis the input ranges from a GB to hundreds of GB. In addition, while the data access pattern of reconstruction is highly predictable, since all the "raw" data have

| Activity | Description | Community | Load | time/job |
|---|---|---|---|---|
| Reconstruction | data filtering | Small | CPU & I/O | 10 hours |
| Montecarlo | data simulation | Small | CPU | 10 hours |
| Analysis | data mining | Large | CPU & I/O | hours to days |

| Activity | Input/Job | Output/Job | Input/Year | Output/Year |
|---|---|---|---|---|
| Reconstruction | GB | GB | 100 TB | 100 TB |
| Montecarlo | None | 10 GB | None | TB |
| Analysis | 100 GB | GB | varies | varies |

Table 3.1: Comparison of different characteristics among three typical computation activities of the DZero experiment. The bottom table focuses on the input/output data size. The numbers represent the order of magnitude.

to be filtered a few times throughout the lifetime of the experiment, the data access patterns of data analysis varies widely, as a few datasets can be accessed over and over again, while others may be almost neglected. All three activities can be run trivially in parallel because of the independent nature of particle physics events. On the other hand, while montecarlo and reconstruction are purely "batch" activities, analysis is run in both interactive and "batch" modes.

These characteristics of high energy physics applications have been used to develop our grid system. Table 3.1 summarizes the order of magnitudes of different characteristics of reconstruction, montecarlo and data analysis for the DZero experiment.

## 3.2 The SAM-Grid Client

The development of a job management infrastructure for the SAM-Grid started as a way to offload the computing of the DZero experiment from the Fermilab computing facilities. In order for this infrastructure to be appealing to the users, at a minimum it had to handle most of the applications commonly used by the community. Instead of developing a framework that would fit generic applications, the SAM-Grid team was pressured to

implement initial prototypes that could be put to use as soon as possible. Priority was given to "production" applications, such as montecarlo generation or data reconstruction, because of their restricted community and predictable data access patterns.

### 3.2.1 A Case for Application-Aware Middleware

Even restricting our system to manage montecarlo generation and data reconstruction only, it was still a challenge to run efficiently jobs with such different characteristics (table 3.1). In order to let the grid organize the usage of the resources efficiently, we had to expose details of the applications to the grid. We present below a few examples, used throughout these sections, where the knowledge of the application helps the grid optimize the resource utilization. We use these examples to address the question of how much application-specific knowledge the grid should have *a priori* to run efficiently.

1. **Database Access**: grid jobs submitted to an execution site are split into multiple parallel instances of the same application by the grid-to-fabric interface (chap. 4). This typically results in dozens to hundreds of jobs starting approximately at the same time and, therefore, accessing key resources essentially concurrently. In practice, not all the services have the same degree of accessibility. In particular for montecarlo generation, the parameters describing what type of physics to generate were accessed from a central database, which initially responded with a "denial of service" to 40% of the jobs. Introducing retrial with randomized exponential back off reduced the final job failure rate to 5%. Despite the reduced failure rate, grid jobs and their retrials increased the load of the database to a point where interactive access was extremely inconvenient (minutes per query). This problem was properly solved by informing the grid of the database access characteristics of the montecarlo application. All the hundreds of jobs submitted by the grid, in fact, were parallel replicas of a single grid job and, therefore, required access to the same input parameters from

the database. The grid-to-fabric interface was enhanced to perform a single database access per grid job, when the job entered the site. The information was saved and reditributed to the parallel jobs by internal cluster transport mechanisms. This solution reduced the "denial of service" failure rate to essentially 0% and still maintained a high availability for interactive database accesses. In conclusion, access to a grid resource (the database) was optimized by instructing grid components (the grid-to-fabric interface) of the characteristics of the application (parallel jobs requiring the same input parameters).

2. **Data Storage Access**: different applications have different typical input data access patterns. Data reconstruction applications begin data processing when a single input file, typically 1 Gigabyte in size, is delivered to the worker node. Instead, data merging applications, used in production operations to concatenate files typically 200 Megabytes in size (sec. 3.2.4), begin processing when multiple "small" input files are delivered to the worker node. Optimizing access to the storage resources with such different regimes is a concern. In the SAM-Grid, applications transfer files from storage services that maintain queues of data access requests. The storage services, in fact, control their load by granting access to the data transfer servers a few requests at the time (sec. 4.3.2.2). Access to a transfer server is granted in the order in which the access request is submitted. When reconstruction and merging applications use the same data queue to access their input, transfer requests for the various input files are interleaved. This leads to inefficiencies, because in real life, on a cluster, reconstruction jobs are one or two order of magnitude more abundant than merging jobs. This means that requests for each input file of a merging application is interleaved with a dozen input files of reconstruction applications. Therefore, before starting processing data, a merging application often needs to wait for these multiple reconstruction transfers to occur, thus substantially increasing its idle time. For a cluster of 900 CPU, this idle time is between one and two hours. This inefficiency can be

reduced by instructing the grid of the input data access characteristics of the application. Knowing the number of required input data files, the data storage service can organize requests from reconstruction applications in a queue different from the requests from merging applications. Thus, a few merging applications only compete amongst themselves for file access, drastically reducing their idle time. Figure 3.2 shows a diagram that illustrates this concepts. In conclusion, as in example 1, access to grid resources (data files) was optimized by instructing grid components (the storage service) of the characteristics of the application (multiple or single input data requirement).

3. **Worker Nodes Allocation**: The grid-to-fabric interface of the SAM-Grid submits multiple batch jobs for every grid job entering the site. How many worker nodes should be allocated for a given application? In general, the fewer batch jobs are submitted, the longer each job runs, and vice versa. There is an acceptable range for the running time of a job. Batch jobs should not run too long to minimize the probability of termination before completion. Jobs are typically terminated because they run beyond the maximum wall-clock time allowed by the local scheduler, or because they are evicted due to a higher-priority job entering the scheduler, or because of hardware failures. On the other hand, batch jobs should not run too short in order to maximize the ratio between running time and setup time i.e. the time needed to prepare the job environment (in the SAM-Grid typically around half an hour). The "suitable" expected running time is managed by the grid controlling the number of worker nodes allocated for running the application. It should be noted that applications may have additional constraints on the number of jobs. These constraints are dictated by considerations on ease of bookkeeping and of recovery after failures. In any case, the number of worker nodes to allocate depends on the type of application. For reconstruction applications, the grid-to-fabric interface allocates a worker node for every file in the dataset specified for the grid job. Given the computational requirements of the recon-

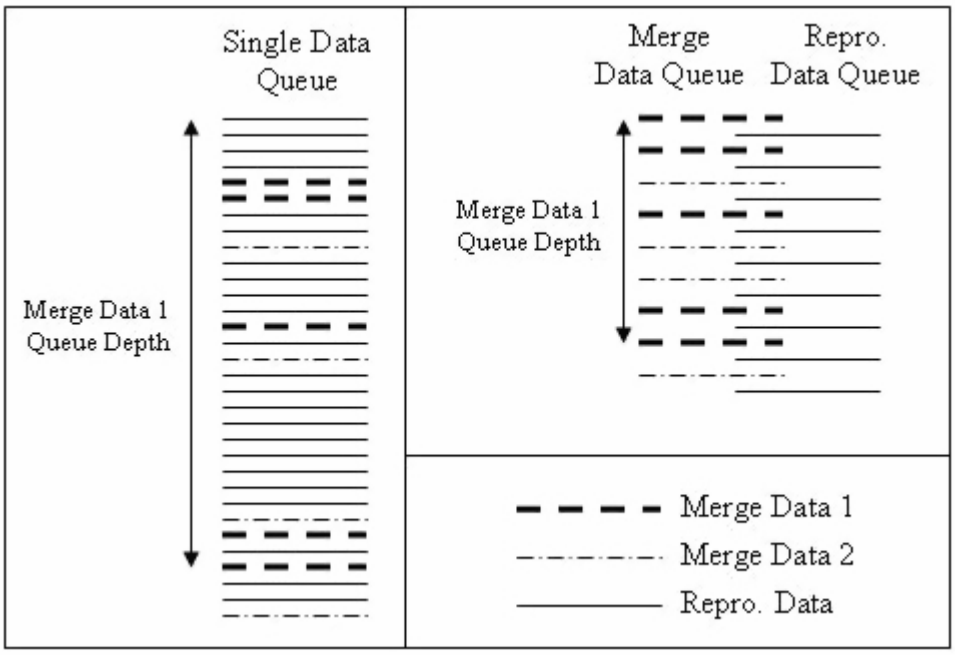


Figure 3.2: A diagram representing queues of requests for file access. On the left, a single queue manages requests from reprocessing jobs (straight lines), and merging jobs (dashed and dashed-dotted lines). Reprocessing jobs are two orders of magnitude more abundant than merging jobs. Merging job 1 needs to access five input files before it can start running (dashed lines, bold for clarity). On the right, requests from merging and reprocessing jobs are managed by two different queues. If access requests are granted one at the time, the queue depth for merging job 1 is much shorter than in the case of the single queue (left diagram). If the data storage server knows the typical data access pattern of the jobs, it can optimize access to the data. The SAM-Grid storage elements have knowledge of the typical data access patterns of each application.

struction application, this approach gives an acceptable running time of a few hours on a modern CPU and eases bookkeeping and recovery operations. For montecarlo applications, the interface computes the number of worker nodes to be allocated by dividing the total number of events to be produced as specified for the grid job by the "optimal" number of events per job. The "optimal" number of events is a parameter configured at the site, considering the speed of the average CPU at the site, the computational requirements of the montecarlo application, and other scheduler constraints (maximum allowed wall-clock time, etc.). In conclusion, as in the previous examples, allocation of grid resources (worker nodes) is optimized by instructing grid components (the grid-to-fabric interface) of the characteristics of the application (computational requirements of the application and other constraints).

4. **Minimal Resource Idle Time**: Grid jobs are often internally composed of interdependent tasks. We call "structured" a job that exposes to the grid its internal structure and let the grid manage the order of execution of each internal task/job automatically (sec 3.2.4). This automation minimizes the idle time between job submissions, thus minimizing the idle time of the resources. In order to decide whether to submit a job, the grid must be able to determine whether the jobs on which it depends were successfully executed. In general, determining the success of a job is a difficult task. In case of complex computational activities, success is generally never defined only by the exit status of the job. To determine whether a montecarlo generation job was successful, for example, the grid has to check the number of events produced by the job by querying a bookkeeping database and compare this number with the number of events originally requested. Success is determined by policy: typically, if more than 90% of the events have been produced, the job is successful. For reconstruction applications the success policy is defined differently: typically a job is successful only if it has reconstructed all the input files, unless subse-

quent recovery jobs fail twice on the same event with the same error, thus exposing a corrupted input file. At any rate, having the grid determine the success of a job is an application-specific task. In conclusion, as in the previous examples, the idle time of grid resources is minimized by instructing grid components (the job management component) of the characteristics of the application (policy defining the success condition).

These examples show how important application-specific knowledge is in the optimization of grid resources. It should be noted, though, that this knowledge must be utilized in concert with resource/service-specific knowledge. For example, only the grid knows in example 1 what database the site should contact, or in example 2 what data queues are available at the certain storage element, or in example 3 how fast the local CPUs are, therefore what the "optimal" number of montecarlo events per job is. In other words, applications should be able to communicate their requirements/preferences to the grid and the grid should be able to communicate its constraints to the application. It turns out that the design of well defined interfaces for such bidirectional communication is a complex problem.

### 3.2.2 A Solution to the Resource Optimization Problem

An approach to define such interfaces is requiring that the applications and the users express requirements and preferences in terms of abstract resource/service-specific quantities. In example 1, the grid could expose to the application an interface that allows the definition of the query to the database. Depending on the application multiplicity, the grid could then decide whether to pool the access to the database or to let the jobs each access it independently. In example 2, the grid could expose an interface to the application that allows the definition of an application name and a number of input files required to start data processing. Depending on these two parameters, the grid could organize data accesses among the available queues. In example 3, the grid could expose to the application an interface that allows the definition of the preferred application multiplicity and its

computational requirements. The grid should then decide whether the preferred multiplicity violates local constraints (e.g. expected execution time beyond wall-clock time limits) and interact with the application to negotiate the final multiplicity. In example 4, the grid should develop an interface to learn how to define the success of the application.

It should be noted that this approach assumes maximum knowledge of the resource/service interfaces from the users and the applications, while the grid does not have any knowledge *a priori* of the applications. This is an ideal assumption for the grid designers that, in practice, may be difficult to maintain. This approach also requires a high level of maturity for the grid interfaces, so that applications could express their requirements to the grid in abstract terms. Until such level of maturity is reached, many optimizations cannot be implemented. Some grids, such as LCG, are taking this incremental approach. Since their operations start in 2007, when the LHC physics program begins, LCG has still a few years to educate its users and to refine its interfaces. In our case, we could not take this approach and we had to program into key grid services some knowledge of the applications. In particular, we had to develop application-specific modules for grid services, in order to encapsulate the knowledge on the supported applications. These are the main reasons for this approach:

- **User Base Expectations**: when we started developing the SAM-Grid, users had to be convinced of the necessity to switch from the central Fermilab system to a distributed computing system. Users were asked to modify their usual way of working and to learn new commands, a new environment, new concepts, etc. For the average user, "learning the way of the grid" was seen as an unavoidable nuisance. If switching to a distributed model had to be done because of financial constraints, the expectation was that at least the transition should be made as easy as possible. This meant that the grid had to expose to the users/applications interfaces as user-friendly as possible. Since the SAM-Grid users think in terms of physical quantities and physics application parameters, the grid interfaces had to be designed

with an application-specific context in mind. In practice, for example 1, this meant to let the users specify a montecarlo request number i.e. an index in a database that holds the physics parameters for a given job, instead of asking them to specify a generic query to the database to retrieve the same parameters. In other words, the concept of "request system", the tool used by the montecarlo user community, had to be implemented in the grid-to-fabric interface for the montecarlo application module. For example 2, the site administrators pre-configure the grid locally with the number of data queues available and with the policies to associate typical application types with these queues. The users only specify the application type for their job and the grid services automatically use the local configuration to organize input retrieval. The concept of application type had to be implemented in the data transport interface. For example 3, the optimal number of montecarlo events per job is configured at the site by the administrators, considering the average computational requirement of the montecarlo application. The users are not required to specify such requirements and only need to declare that the application is of montecarlo type. The concept of physics event had to be implemented in the grid-to-fabric interface for the montecarlo application module. For example 4, success policies had to be pre-loaded in the job management components. In conclusion, application knowledge has been implemented in key grid services. This knowledge is used by the grid to optimize resources for a given type of application. Users configure these optimizations using both application-specific and resource/service-specific characteristics for the job. The SAM-Grid shields its user community from the complexity of purely resource/service-specific interfaces.

- **Incremental Experience Building**: the SAM-Grid optimizes resource usage exposing interfaces that are both application-specific and resource/service-specific. The optimizations are the results of years of experience running a grid production system. Our understanding of the required optimizations has been incremental and came always as

the result of the study of a particular use case in running a specific application. It was natural for us to organize the optimization strategies according to application types. Subsequent iterative refinements of the optimization strategies were also simplified using an application-oriented organization of the grid architecture. In addition, in order to eliminate any application-specific knowledge from the grid, a big effort of abstraction on the application use cases and optimizations is necessary. For new applications, an understanding of the necessary optimizations is generally missing, which makes it difficult to abstract the purely resource/service-specific interfaces. In other words, this approach would make it more difficult to introduce new applications and their optimizations. Such generalizations would also require a higher level of understanding of the grid technologies and interfaces from the users, which, at the moment, is a disincentive to use the grid.

- **Interface Definition Process**: high energy physics applications generally require a careful preparation of the job environment before running. This preparation includes the definition of dozens of environment variables, the creation of complicated file system directory structures, the placement of execution, configuration, and input files in them, the instantiation of monitoring processes that reduce the impact of rare bugs that lead to infinite loops, etc. The environment preparation is such a complicated process that application management middlewares have been written to help with it, for example Runjob [119]. It is often the case that the application management middleware interacts with the grid on behalf of the application. Every time a grid interface is modified, the developers of the application management middleware have to update the software. Conversely, the grid interacts with application through the application management layer. In case these interfaces change, the grid software needs to be updated. Because different groups of people work on the two projects, the process of changing interfaces must be though through carefully and modifications must be planned and negotiated. Moving from application-specific grid in-

terfaces to purely resource/service-specific interfaces requires a long process for which there is currently little incentive.

### 3.2.3 The SAM-Grid Job Description Language

The SAM-Grid components know *a priori* the applications supported. These components expose application-specific interfaces in addition to resource/service-specific interfaces. This approach allows the optimization of grid resources to efficiently execute jobs.

Users describe the characteristics of a job using a Job Description Language (JDL). The JDL contains terms typical of the various applications as well as of resources and services. The characteristics of the job described in the JDL are translated in directives to the appropriate services on the grid. We believe that the language is defined through an appropriate balance of terms specific to applications and to resources/services. A description of the SAM-Grid JDL and of the underlying directives to the grid services follow.

The SAM-Grid JDL is a list of attribute/value pairs. The terms specific to resources/services, application-neutral, include, for example, the name of the local directory where the user software is (*input_sandbox*); the name of the SAM station to be used (*sam_station*); the generic set of requirements expressed in terms of advertised site characteristics (*requirements*). The terms specific to applications change depending on the application type. When running a montecarlo application, for example, real detector events, called in this context "minimum bias", are used in the process; the attribute *minimum_bias_dataset* identifies the name of the set of files to be used for minimum bias. Again for montecarlo, DZero users submit their requests for production of events to a request system; the id of the request is processed by Runjob, the application management middleware of the experiment, to organize the appropriate execution sequence for the applications (*runjob_requestid*). The full list of attributes and their semantics can be found in the SAM-Grid manual [120].

The SAM-Grid client software is designed in two distinct layers. The top layer, closest to the user, exposes the interface specific to the high en-

ergy physics domain. The bottom layer, which consists of standard middleware from the Condor-G system, implements the low-level job management mechanisms. In particular, for job submission, the top layer corresponds to the SAM-Grid Job Description Language (JDL) interpreter and the bottom layer to the native Condor-G submission client. This section focuses only on the JDL interpreter.

The main responsibility of the JDL interpreter is to translate the description of the job provided by the user into a low-level set of directives to the services of the grid. These directives can be application-specific or resource/service-specific. The directives can be categorized based on what grid services they affect:

- Directives sent to the interpreter itself: these have an effect in particular on the creation of the software archive containing the user executable, libraries and configuration files. Directives for this purpose are application-neutral. In jargon, this archive is called the "user input sandbox". It should be noted that in the SAM-Grid model the user sandbox is, in principle, only part of the software required to run the overall application. The rest of the software is retrieved via the data handling system and dynamically deployed by the sandboxing fabric services (sec. 4.3). Another example of this type of directives instructs the interpreter to run consistency checks of the job description. This example is an application-specific directive, because different applications require different checks for consistency. For montecarlo, the information in the request system is compared with information specified by the user. This type of check, for example, simply does not apply for reconstruction. These checks are important because the typical latencies of a grid imply that trivial mistakes in the value of the parameters are detected by the infrastructure hours after the job submission and they generally result in unrecoverable failures. The consistency checks are essential to implement a fast fail submission mechanism.

- Directives sent to the Resource Selection Service: these affect the logic used by the resource selector to associate a job with an execution site.

An example of a typical logic used for our prototypical implementation of data analysis applications gives priority to those sites that have already cached the highest percentage of the data requested by the job. Another typical logic is random site selection (sec. 3.4.3).

- Directives sent to the submission site services: directives in this category affect the algorithms that manage structured job. A typical example configures the behavior of structured jobs composed of a montecarlo and file merging applications (fig. 3.7). One directive sets the maximum number of times the montecarlo application should be resubmitted in case of failures. Another directive defines the percentage of events that the montecarlo job should produce in order to consider it successful. These directives are useful to address the optimization problem raised in example 4 sec. 3.2.2. Structured jobs are discussed extensively in section 3.2.4. Another directive in this category is the configuration of the email address to which notification should be sent after job completion. Also, directives in this category can configure the logic for resubmitting/rematching jobs in case of failure (sec. 3.3).

- Directives affecting the fabric services at the execution site: an example of application-neutral directives in this category are aimed at recreating the job execution environment. In particular, they can program the dynamic product installation mechanism, declaring specific products as necessary to the job (sec. 4.3). Application-specific directives include the montecarlo request id. This is used to run a single query to the database at the site in order to gather the application input parameters (example 1 sec. 3.2.2). It should be noted that gathering these input parameters at the client may slow down significantly the user interface. Directives to the fabric services are often used in conjunction with site-specific configuration to coordinate and optimize resources/services when running specific applications (examples 2 and 3 sec. 3.2.2).

- Directives to the data handling system: these include, for example,

```
job_type = dzero_reconstruction

station_name = imperial-prd
sam_experiment = d0
sam_universe = prd
group = dzero

input_dataset = dayset-2004-08-18-196489-0
jobfiles_dataset = d0repro_jobfiles_p17.02.00_2
d0_release_version = p17.02.00

check_consistency = true
instances = 1
```

Figure 3.3: A SAM-Grid job description file for the reconstruction application (*job_type*). The second group of attributes from the top defines the remote cluster where to run the job, identifying a specific SAM station. The third group defines the input file dataset (*input_dataset*), the application binaries (*jobfiles_dataset*), deployed to the worker node via SAM, and the application version (*d0_release_version*). The fourth group instructs the interpreter to submit a single instance of the job (*instances*) and to check the consistency of the job description (*check_consistency*) e.g. by checking that the files in the binary dataset (*jobfiles_dataset*) include the code for the application version specified.

what dataset the job requests (sec. 1.3) or what physics group is accountable for the "cost" of the data caching.

- Directives to the application itself: these are passed as arguments to the application or to wrappers around the application.

Figure 3.3, 3.4, 3.5, and 3.7 show examples of job description files. The types of jobs supported are montecarlo production, data reconstruction, data merging and "structured" jobs.

Other groups have developed job description languages [121, 122], similar in many aspects to the SAM-Grid JDL. One of the most mature JDLs is the one developed for the European DataGrid [122]. The most substantial difference with the SAM-Grid JDL is that the DataGrid language is

```
job_type = dzero_monte_carlo

station_name = samgfarm
sam_experiment = d0
sam_universe = prd
group = dzero

runjob_requestid = 20214
jobfiles_dataset = hynek_jes_group_15400
d0_release_version = p14.07.00
runjob_numevts = 12750
events_per_file = 250

minbias_dataset = overlapset_mcp14_cteq5l-tuneA_simulated
phase_dataset = CKKW_Rand_1.W+4run4.1.io
phase_skip_num_events = 12250

check_consistency = true
instances = 1
```

Figure 3.4: A SAM-Grid job description file for the montecarlo application (*job_type*). The application gathers the processing details from the request database (*runjob_requestid*). Some of the parameters expressed in the database, such as the software version (*d0_release_version*) or the number of output events (*runjob_numevts*), can be overridden with the appropriate JDL attributes. The number of events per output file (*events_per_file*) is a site-specific value that can be overridden. The fourth group of attributes define the input to the application in terms of SAM dataset names (*minbias_dataset* and *phase_dataset*) and how to seek the relevant events within these files (*phase_skip_num_events*). The other attributes have been discussed in figure 3.3

```
job_type = dzero_merge

station_name = d0karlsruhe
sam_experiment = d0
sam_universe = prd
group = dzero

#merge_dataset_name = tmb_output_12403_2
merge_dimension_query = global.requestid=12403 and data_tier=thumbnail

jobfiles_dataset = copyd0om_jobfiles_merge_p14.05.01_1
d0_release_version=p14.05.01

check_consistency = true
instances = 1
```

Figure 3.5: A SAM-Grid job description file for the merging application (*job_type*). The files to merge can be expressed as a set of constraints in the metadata catalog of SAM (*merge_dimension_query*) or as a SAM dataset name (*merge_dataset_name*). The # sign is a comment. The other attributes have been discussed in figure 3.3

completely resource/service-oriented. In other words, it does not provide facilities to assist with any specific families of applications. The DataGrid infrastructure is scheduled to go in production in 2007, when the LHC experiments will start taking data. Most of the applications that will run on the DataGrid are not finalized and the community is not yet worried about resource optimization. As noted, in our experience, knowledge of the application is required to achieve an acceptable job efficiency.

Other relevant differences depend on the different service architectures of the two grids. For example, the SAM-Grid JDL relies on SAM to express data handling needs. In particular, the SAM system transparently moves required data to a location close to the user. Instead, the DataGrid JDL refers directly to EDG Storage Elements and transfer protocols. Another example is the EDG attributes that invoke services not available or not directly programmable by the user in SAM-Grid, such as accounting, credential management, and checkpointing.

Despite the differences, many attributes are common between the two languages, if not syntactically at least semantically. Both JDL have attributes to identify the Virtual Organization of the user. Both can express job requirements and rank sites in terms of advertised resource characteristics. In particular for ranking resources, both grids have developed mechanisms to involve the cost of input data transfer in the selection criterion (*GetAccessCost function* for EDG, *sam_rank_data_overlap function* for SAM-Grid).

Other attributes simply apply to one grid and not to the other. For example, in EDG a job can be classified as one of the following four types: normal (batch), interactive, checkpointable, MPI. The SAM-Grid, instead, always runs jobs in batch mode and reserves the JDL keyword *job_type* to identify application families. Running jobs on the grid in interactive mode is a problem that the SAM-Grid has not investigated yet.

### 3.2.4 Structured Jobs

High energy physics computations are often the result of multiple applications run in a specific order. The input of one application, in fact, is often the output of another. This relationship determines a hierarchy of dependencies among applications. Thain et al. call these types of job "batch-pipelined workloads" and analyze six use cases, including high energy physics, in [123].

A typical pipeline for the DZero experiment is the production of montecarlo events. The internal structure of the pipeline is a "chain", that is, the output of an application is the input of the next application in the chain, and so on. More specifically, particle events are first randomly generated by an application; a second application simulates how the detector would see these events; a third one refines the simulation, adding trigger and data acquisition efficiencies and other physical effects, such as pile up; finally the events are "reconstructed" i.e. translated into a format convenient for the analysis framework. Only the final output of the computation is worth keeping. The intermediate results are only recorded with the data handling system as "virtual" data i.e. files that do not exists physically in the system, but whose meta-data are saved for bookkeeping purposes.

We concentrate on the problem of running this type of applications on the grid [124]. Even before the grid era, DZero relied on an application management middleware, Runjob [119], to manage the workflow of pipelines such as montecarlo production. Runjob prepares the environment of each task in the chain, feeding the output of an application as the input of the next. For montecarlo production, Runjob executes all the tasks in the pipeline on the same node. This approach is most efficient, as the workflow of the data is contained within the processing node and there is no latency for data stage-in/stage-out.

On the other hand, the typical montecarlo request from the users consist of the production of 20000 simulated events. If all the events were to be produced on a single CPU, the task would take, on average, forty days (figure 3.6A). This is too long for most clusters to guarantee a reasonable chance of success: a job is considered "long" if it runs for a few days. To

be on the safe side, ideally we would not want to run a job for more than a day.

Running each of the four tasks of the pipeline separately and saving the output at the end of each task would not help with this problem either. The average running time per task would be, in fact, ten days (figure 3.6B). The solution is running the whole chain completely producing a subset of the events each time. This approach is also immediately parallelizable (figure 3.6C). But what is the "optimal" number of events that a single job should produce? It turns out that this number is the result of a trade-off between the job running time, which should be ideally around a day, and the size of its output, which should be around one Gigabyte.

The reasons for the one Gigabyte output size are the following. From a data handling point of view, it is more efficient to handle a few large files, than a lot of small files. In particular, accessing a mass storage system causes large latencies if a dataset is fragmented into too many files. Another cause of inefficiency in the data handling is the interaction with the metadata catalog. A dataset organized in a lot of small files requires a large number of database accesses as compared to the same dataset organized in a few large files. On the other hand, files must be smaller than 2 Gigabytes, the maximum file size for most operating systems. CDF and DZero have determined independently that a good trade-off between size and access efficiency is 1 Gigabyte.

On the other hand, a job that produces an output of 1 Gigabyte may need to run for much longer than the canonical one day. A montecarlo pipeline produces on average an output of 70 Megabytes in twelve hours. In order to produce a 1 Gigabyte file, the application should run on a single node for one week, again too long. These two conflicting requirements (job running time and output size) do not affect only montecarlo production. Data reconstruction applications are very CPU intensive and produce small output files even for large input files. For ease of bookkeeping and recovery from failures, each job is given a single file to reconstruct. Yet, reconstructing a 1 Gigabyte file takes about a day and produces on average 200 Megabytes of output, again much smaller than the canonical 1 Gigabyte.

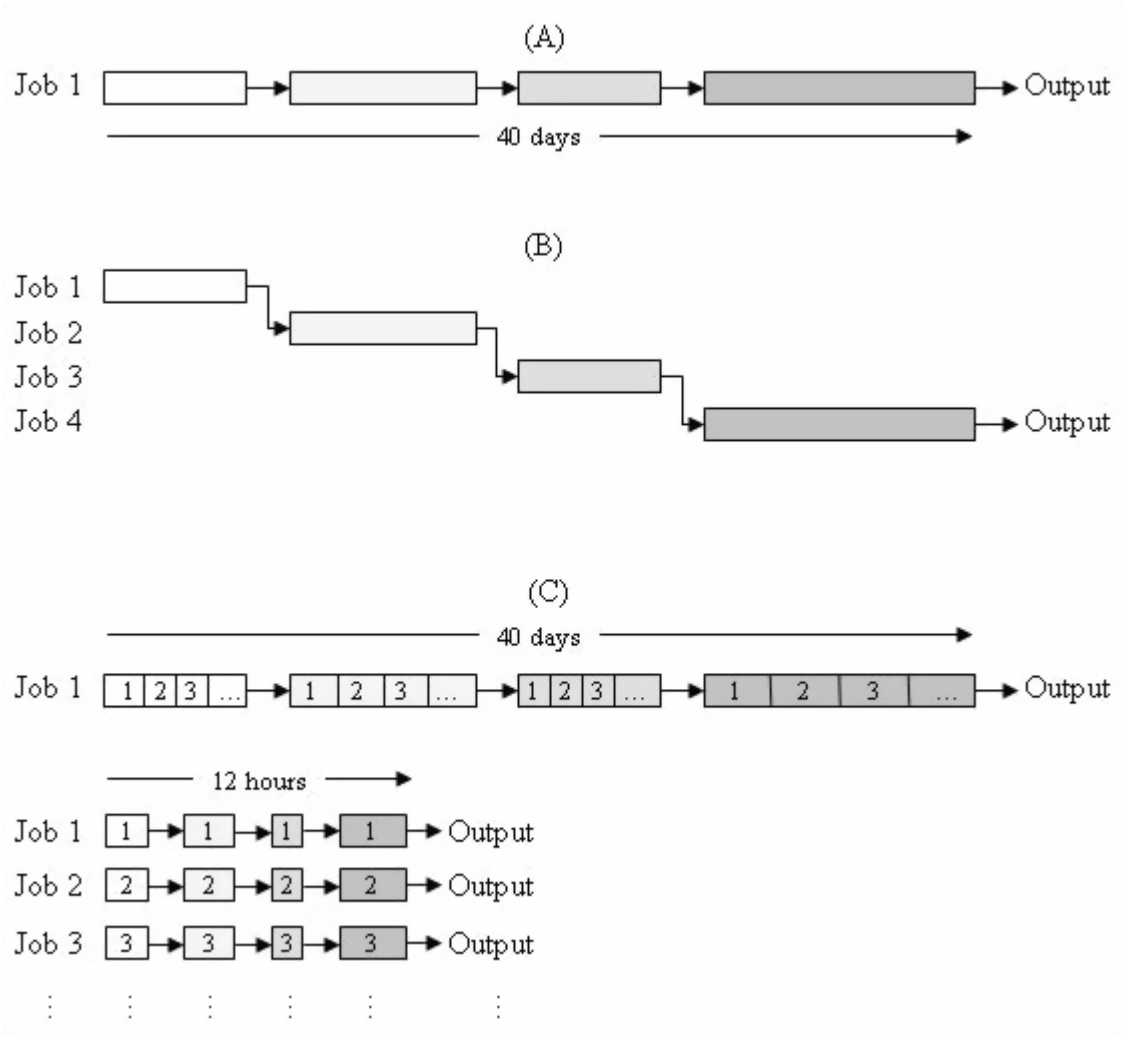


Figure 3.6: (**A**) A montecarlo request, consisting of four applications represented by the gray bars, form a pipeline job. The typical montecarlo request would run for forty days on a single commodity CPU: this is too long for most clusters to guarantee a reasonable chance of success. (**B**) The same pipeline is run as four separate jobs. Each job would run on average for ten days: this is still too long. (**C**) The request is parallelized into hundreds of "small" pipelines, each running for less than a day. The running time is "right", but the output of each job is too small for efficient data handling (much less than one Gigabyte). The output files from all the jobs must be merged together.

To reconcile the data handling needs with the ideal running time of less than a day, we have developed an application whose only task is to merge files. Montecarlo production and data reconstruction jobs are run in parallel at the worker nodes of the cluster. These jobs are all part of the same computational request, for example, the production of 20,000 montecarlo events or the reconstruction of an input dataset of one hundred files. The jobs are run in parallel for one day and their "small" output is stored from each worker node on a temporary data storage (typically a disk), considering it as an intermediate result. Once all the jobs are finished, the merging job is run on a worker node. The job gathers all the "small" input from the same computational request and produces output files of 1 Gigabyte in size. These files, which have the "right" size, are permanently stored on a mass storage system.

This task is greatly facilitated by storing the intermediate results in the SAM system. Since all the files are thoroughly catalogued, the process of selecting what files to merge simply consists of creating a dataset definition i.e. a query in the SAM metadata catalogue. The files are stored physically on a storage system close to the cluster. Ideally, this storage system should provide some standard data handling interface, such as SRM [125]. In practice, it is almost always a disk on a machine that provides some data movement service. We call such storage system a *durable* location, to hint that it holds files that are not used for scratch, since they survive the life time of jobs that produced them, but are not stored there permanently. Storing the intermediate results in the SAM system also enables the transparency of data access. In other words, the merging job can be run anywhere on the grid, as SAM will take care of the transport on behalf of the application.

It should be noted that this processing model is another pipeline consisting of two applications: the production job (the parent) and the merging job (the daughter). In contrast to the montecarlo production pipeline described above, this pipeline is executed on the grid. In other words, in order to execute the new pipeline, the grid needs to know its internal structure. We call a job that exposes to the grid its internal structure a "structured" job. It should be noted that the montecarlo production pipeline is not a

structured job, as it is treated by the grid as an atomic task.

More generally, the relationship between jobs can be conveniently modeled as a Directed Acyclic Graph. Cycles can be introduced in case of failure, by automatically rerunning the failed job. Even with the simple production and merging pipeline described above, we have observed two operational problems in the manual handling of structured jobs:

1. Operational mistakes: production jobs tend to run for many hours or days. In addition, system crashes, such as software or hardware failures, require operational intervention for recovery. The long processing time, together with the recovery procedures, leads to human errors.

2. Low production efficiency: since a job stays in the grid system for days, it sometimes finishes at times when operators are not available to check its success and to submit the dependent jobs. These dead times contribute to lower overall production efficiency.

The solution to these problems is the development of an infrastructure that handles automatically structured jobs (requirement 6 chap. 3). The user interface should allow an operator to declare the following characteristics of the job at the time of submission: (1) all the phases of the job and their relationship; (2) the checks to determine the success of each jobs and the policy for the resubmission of failed jobs. A typical example of such policies is the number of times that the job should be resubmitted before considering the failure un-recoverable without human intervention.

This automatic system is in practice difficult to implement. In order to run a computation as a structured job, in fact, the details of the operations must be completely sorted out. The operational checks that are normally run after every job in the structure must be so clear that they can be automated. This automation is often difficult to achieve in some cases, typically because of the high number of possible error conditions. In addition, every application generally requires a different type of check. This means that it is sometimes difficult to generalize the checks without having a grid with *a priori* knowledge of the application (sec. 3.2).

Various groups interested in high energy physics make also use of structured jobs [42, 126]. It is interesting to note that these groups, as well as the SAM-Grid, have implemented structured jobs using the same middleware, the Direct Acyclic Graph job Manager (DAGMan) [127]. DAGMan is a workflow management layer on top of the Condor middleware. An input file to DAGMan defines the dependencies among the jobs. This file is used by DAGMan to compute the order of job execution. Each job is described by a condor job description file, which specifies the job executable, arguments, input and output, etc. In addition, the execution of each job can be preceded and followed by custom pre/post-processing scripts. Typically, pre-processing scripts are used to stage the job input files. Such scripts are not used by the SAM-Grid, as jobs use the SAM data handling system for their data handling needs. Post-processing scripts can be used to determine whether the job was successful or not. These are heavily used by the SAM-Grid, as they are the key for the automation of the structured job execution. The number of times a job should be resubmitted before considering the job failed is expressed in the DAGMan input file. It should be noted that the ability of resubmitting jobs upon failure increases the overall fault tolerance of the system.

Internally, DAGMan implements job scheduling running a "master" job at the submission tier. The master job has knowledge of the dependencies of the jobs in the DAG and has access to all information necessary to submit each job. The master job enforces job dependencies by submitting the jobs of the DAG to the scheduler at the appropriate time. Pre/post-processing scripts are also run at the submission tier by the master job.

Being a general tool, DAGMan offers features that are not currently used by the SAM-Grid. On the other hand, the SAM-Grid could benefit from features that are not yet implemented in the tool. The most notable of the features currently not used by the SAM-Grid is the automatic recovery of DAG jobs. In case some jobs fail, this feature allows the re-submission of only the failed jobs and its dependencies. This is mostly of interest for complicated structures and it is not currently necessary for the types of structured jobs run on the SAM-Grid. On the other hand, the SAM-Grid

would benefit if DAGMan were integrated with a multi-tiered submission system, a feature that is not available yet. In more detail, DAGMan currently requires that the submission tier, i.e. the server that submits jobs to the remote resource, and the client tier be on the same machine. This limitation stems from the fact that the client cannot currently gather and ship all necessary information to the submission server. This information includes all descriptions, executables, and input files of the jobs in the DAG, as well as the DAG description itself. A multi-tiered architecture for the management of DAG jobs would be in line with the overall architecture of the SAM-Grid. In order to run structured jobs, the SAM-Grid accept the limitation of running some client and submission tiers on the same machine.

The SAM-Grid client provides a level of abstraction on top of DAGMan. Users declare structured jobs using a single SAM-Grid job description file that includes the relevant attributes for the jobs in the SAM-Grid JDL. Currently, only chains of applications are supported by the SAM-Grid client. Structured jobs are typically used to run montecarlo production and output merging applications. Figure 3.7 shows a real example of such a job.

## 3.3 The Job Submission Service

When we started working on the job management infrastructure of the SAM-Grid, the size of our user community was a parameter important in our design. The DZero experiment had around 700 members and CDF around 800. For DZero, every active user had an account at Fermilab on the d0mino super-computer, which was deemed expensive to maintain and operate. The effort to administrate such a large system for a growing number of active members was not negligible. On the other hand, most users already had personal desktops at their home institution, or laptops. If the client software of the job management infrastructure could be operated from these computers, the effort of administering the central system could be, at least in part, relieved.

As the number of client machines grew in our architecture, the probability that one of them was down during the life time of a job was also growing.

```
job_type = structured
job_structure = dzero_monte_carlo, dzero_merge

station_name = samgfarm
sam_experiment = d0
sam_universe = prd
group = dzero

runjob_requestid = 12446
d0_release_version= p14.05.01
jobfiles_dataset = adi_jobfiles13

minbias_dataset = ccin2p3_minbias_dataset

monte_carlo_efficiency = 90
monte_carlo_retries = 3

instances = 1
```

Figure 3.7: A SAM-Grid job description file for a structured job (*job_type*), consisting of a chain of montecarlo production and merging jobs (*job_structure*). The last group of attributes are parameters passed to the post-processing script of the production job. This defines the success of the production job if 90% of the requested events have been produced (*monte_carlo_efficiency*). The job can be resubmitted 3 times (*monte_carlo_retries*) to reach this "quorum", before considering the whole job failed. It should be noted that none of the typical attributes for merging jobs are present (see figure 3.5). The dataset to be merged, in fact, is determined using the global job id of the production jobs. The other typical attributes are common with the montecarlo job. The second, third, fourth groups of attributes have been described already in figure 3.4.

Yet, the architecture had to provide the same level of reliability as a central system. In other words, the jobs submitted through this system should not require the availability of the client machines in order to succeed. If this were achieved, we could also support the installation of the client software on a laptop. Therefore, a question that the SAM-Grid had to address was how to manage jobs without requiring continuous network connectivity for the client software. The solution that we adopted was an agent, called the submission service, that would act on behalf of the user for the management of grid jobs. The clients would interact with this service when necessary, and then they could disappear from the network.

It is important that the submission service is reliable, since multiple clients depend on it for the successful management of their job. The service has to treat each request persistently, so that temporary down time of the service does not jeopardize the running jobs. In addition, it is necessary to clarify the authority that the service has over the jobs. For example, could the service resubmit failed jobs? Should the service receive the output of the jobs and keep it for the user? If so, for how long?

The submission service is the second tier of the job management component, interacting with both the client software at the first tier and the execution site services at the third tier. The designed multiplicity of the submission service is about one per nation. As figure 3.1 shows, multiple clients interact with an instance of a submission service[1].

This component has four main responsibilities

1. It maintains the queue of grid jobs submitted there, holding the job description as well as the user input sandbox. The information is stored persistently and the status of the service and of the jobs can be recovered after a service shutdown (requirement 3, chap. 3).

2. It interacts with the Resource Selection service, providing upon request the description of the pending jobs and receiving a recommendation

[1]In the SAM-Grid deployment of August 2005, the nations that offer a submission service are the United States (at Fermilab and at Langston University, Oklahoma), Canada, France, and the Czech Republic. Clients in the UK and Germany typically interact with the Fermilab's submission site.

on where each job should be submitted (sec. 3.4).

3. It schedules the job at the site recommended by the Resource Selection service. The main protocol used for the interaction with the remote resources is the Globus Resource Allocation and Management (GRAM) protocol (sec. 1.1).

4. It is a mediator of the interactions between the user and the remote resources for the management of the jobs, when the user is online. Conversely, it acts on behalf of the user, when the user is offline (requirement 2, chap. 3). In particular, the submission service maintains up to date the status of the job, can in principle resubmit the job to the grid resources in case of failure (requirement 5), and receives the output sandbox from the job upon completion (requirement 3).

The submission service has been implemented using the "schedd" daemon of the Condor-G infrastructure. The Condor software had to be modified to allow the separation between client software and the schedd. In addition, the daemon had to be appropriately configured to meet the policies of the experiment, as described hereby.

The handling of the job output, or, more precisely, of the output sandbox, was a problem that required custom configuration. In the SAM-Grid model, the output sandbox consists of "small" user files and meta-processing information, such as the standard output and error streams and the application log files. In other words, the output that is of little interest to the community as a whole and does not need to be catalogued. We make this distinction because the main results of the computation, often many GB in size, are catalogued and handed over to the data handling system. The information in the output sandbox is of fundamental importance for troubleshooting the system as well as the application, despite the fact that it is not permanently stored nor catalogued.

We decided that the submission service would be responsible for keeping the output sandbox upon job completion. The user can download the sandbox from the submission site via the web. If the output is not retrieved, the

submission service can delete it, according to specific policies. Typically, the output is held for a week, before disposing of it. Such policies are programmed in the job description and can, therefore, be on a per-job basis or global.

Other grids adopt different approaches for dealing with the job output. A popular model requires that the users specify a location to store the output. This model is adopted with some variations by LCG as well as by the CDF distributed Analysis Farms (CAF) [118]. In the case of CAF, the user is responsible to check that the chosen storage accepts the user credentials and that it has enough space to store the output. If these conditions are not true, generally the system buffers the output for some time at the execution site. In practice, today the CAF system is fully integrated only with kerberos, instead of more common grid standards, such as the Globus Security Infrastructure, and the storage system of choice is usually the user's home area. Within these restrictions, though, most users consider that the convenience of finding the output directly in the user's home area overcomes the responsibilities that the model imposes to them. On the other hand, we believe that the SAM-Grid model offers a reasonable trade off between user convenience and responsibilities, as it requires the user to download the output, but does not require the users to be responsible for the availability of the storage system.

We conclude this section by introducing the problem of fault tolerance on the SAM-Grid. In the context of the SAM-Grid, this problem is a topic for future research. Fault tolerance has been studied in the context of other grid systems by various groups [129, 130, 131]. These are general guidelines of how it would be possible to implement fault tolerance within the SAM-Grid framework. The submission service could increase the fault tolerance of the infrastructure if it were granted the authority to resubmit failed jobs on behalf of the user. On the other hand, the submission service should be taught how to distinguish recoverable from non-recoverable failures.

We distinguish between three possible actions of the submission service:

1. resubmit the job selecting a different resource i.e. rematch the job

2. resubmit the job to the same resource, optionally waiting a period of time

3. consider the job failed

In general, if a job fails because the resource selector does not know of local problems at the resources or has stale information, the job should be resubmitted. Whether to resubmit or to rematch the job would depend on the type of failure and on how permanent the failure is expected to be. For example, if accessing gateway services fails because of a network timeout, it is probably best trying to resubmit to the same resource. The gateway node, in fact, may be busy processing other requests, but otherwise generally available. Instead, if submitting the job fails because the submission service cannot establish the output streams with the gateway node, the job should probably be rematched. Such failure, in fact, generally occurs when a filter blocks the network (firewall configuration) and typically requires administrative intervention to be solved.

On the other hand, a typical example for considering the job failed is when the job credentials are expired. In this case, the submission service should not try to resubmit the job, as it would only generate traffic without any chance of success. Knowledge of application would also help deciding on the resubmission strategy. Some applications, in fact, are non-reentrant by design. This means that they cannot be submitted multiple times without undergoing some operational recovery procedures. For example, this is the case of reprocessing jobs. Before resubmitting a grid job, operational scripts should analyze what files in the original dataset were not successfully processed and create a recovery dataset containing them.

As of today, the SAM-Grid team has only experimented with some very simple resubmission algorithms that do not clearly distinguish among failure conditions. These algorithms penalize previously matched resources in case of resource rematch, as the history of the matches are recorded in the job description. This solution to increase the fault tolerance of the SAM-Grid was never finalized. Categorizing the types of failures and teaching the submission service how to distinguish them is still an open problem.

## 3.4 The Resource Selection Service

When we started the design of the SAM-Grid job management component, a principal goal was offloading the computing resources at Fermilab by making remote resources accessible. The number of DZero collaborating institutions was 78, of which only a few dozens provided computing clusters of a medium to large size (dozens to hundreds of machines). Rather than considering each machine individually, the target of the SAM-Grid scheduling was the cluster. The grid, in fact, had no direct scheduling capability over resources at the granularity of the single machine. In addition, the aggregation at the level of the cluster could shield grid services from an excess of details, irrelevant for most practical purposes (sec. 2.1.6). Maybe even more importantly, users think in terms of sites or clusters, when deciding where to submit their jobs.

At a minimum, the job management component had to be capable of submitting jobs to a cluster chosen by the user. Users, in fact, tend to submit jobs to the cluster hosted by their home institution. The incentive to do this is that the computing power utilized at the institution is counted as a credit toward the yearly due to the experiment for operations. In order for the submission site to submit jobs to a named cluster, various details of the resource must be available, such as the "Globus URL", the entry point of a resource accessible via the GRAM protocol (sec. 1.1). In addition, in order for the job to have a chance of success, other resource-specific characteristics must be made available in the job environment, such as the name of data handling services and their naming service, the entry point of the monitoring service (XML database URL: section 2.2.3), etc. It is therefore necessary that the grid provides an information service, where all these attributes could be found.

Ideally, the job management infrastructure should be able to interface directly to the information system. Some grids [88], for example, provide the information on the resources via a web interface and leave to the user the responsibility to fill in the relevant attributes by hand, for example by cutting and pasting the information to the job description file, creating a *concrete* job request. We believe that this is too demanding of the user and

| Concrete Representation | | |
|---|---|---|
| GatekeeperUrl | = | gateway.machine.fqdn:2119/jobmanager-DZeroLong |
| XMLDatabaseUrl | = | http://xmldb.machine.fqdn:7080/Xindice/ |
| ... | | |

| Abstract Representation | | |
|---|---|---|
| GatekeeperUrl | = | $$(GatekeeperURL) |
| XMLDatabaseUrl | = | $$(XMLDatabaseUrl) |
| Requirements | = | TARGET.Site == MySite AND |
| | | TARGET.SchedQueue == DZeroLong |
| ... | | |

Table 3.2: The Condor JDL representation of an abstract and a concrete job descriptions. In the concrete description, the user must enter the exact value of each attribute in the job description. In the abstract description, the user expresses requirements using abstract resource attribute names (*TARGET.* syntax, section 3.4.3), in order to select a resource. The concrete values are substituted by the resource selection service using the information system (*$$* syntax). The abstract representation is more user friendly.

that the grid should provide a service that would automatically fill in the details, given a few mnemonic *abstract* attributes of the targeted cluster. Table 3.2 shows an example of an abstract and a concrete job descriptions, using the Condor job description language.

The problem that the SAM-Grid had to address was how to translate an abstract job description into a concrete job description. This translation service relieves the user from the tedious work of looking up static resource details on the information system. But what about dynamic conditions, such as the current load to the cluster or the data handling components at the site, the network traffic, etc. ? More in general, the user may not care on what cluster his/her jobs execute and would rather have the grid select the "best" resource on his/her behalf (requirement 4 chap. 3). Clearly, the "best" resource selection criterion depends on what metrics we want to optimize. In our experience, the most popular metric among users is time to

completion of the job. On the other hand, administrators care about the optimization of site parameters, such as the minimization of network or storage system bandwidth. Finally, it should be noted that the resource selection mechanism depends on the application type. CPU intensive applications, for example, may not need to care about the load of the data handling services at a site, while data intensive applications do. In summary, the question that we asked ourselves in the context of SAM-Grid was how to select the optimal resource for a given application in order to optimize metrics relevant to our stakeholders.

In the rest of this section we explore these questions further by presenting relevant aspects of the literature on resource selection. In the next subsections, we analyze the problem of job and data co-location and we describe the SAM-Grid framework, used to implement resource selection, answering at least in part the questions introduced above. We also describe how the SAM-Grid has prototypically implemented a mechanism for job and data co-location to run analysis jobs.

Many groups have worked on the problem of resource selection on the grid. Buyya, Chapin and DiNucci [132] study different architectures for resource management, comparing models that organize resources and services in a hierarchy with models inspired by the economic principles. A more comprehensive work is proposed by Krauter, Buyya and Maheswaran [133], who present a taxonomy of the resource management systems using a dozen characteristics. The three major types of Grids identified are Computational Grids, Service Grids and Data Grids. In a Computational Grid, applications are executed in parallel on multiple machines as if the aggregate system were a supercomputer. Storage management is provided via specialized infrastructures. A Service Grid is a system that provides aggregate services that are not provided by a single machine. Typical examples are systems that connect users and applications into collaborative workgroups and infrastructures for realtime multimedia applications. A Data Grid integrates grid level data management services with computing resource management services. This makes data grids more complex than computational grids, a reason why the research on data grids is not as advanced. The focus of a

data grid is generating new information by processing data from distributed repositories. The SAM-Grid is an example of a Data Grid.

For computational grids, the literature on resource selection and scheduling algorithms is conspicuous [134, 135, 136, 137, 138]. Bruno, Coffman, and Sethi study the finishing time properties of algorithms for scheduling $n$ independent tasks on $m$ nonidentical processors [139]. They show that the simplified problem of scheduling a restricted set of tasks on identical processors is NP-complete. Fujimoto and Hagihara [140] measure the performance of a few classical algorithms and compare their relative performance. Buyya, Stockinger, Giddy, and Abramson [141] concentrate on economical scheduling models.

In addition to the theoretical work, various groups have implemented resource selection middleware for computational grids [29, 32, 142, 143, 144], and for hybrid computational and service grids [145, 146, 147, 28]. Buyya compares these technologies in his Ph.D. thesis [148], investigating the advantages of an economical model using the NimrodG system.

In the case of the SAM-Grid, the resource selection service gives recommendations to the submission sites on how to match jobs to execution sites. In other words, in the architecture of the job management component, this service multiplexes the jobs queued at the second tier to the third tier. As for the submission and client site software, the SAM-Grid implementation of the Resource Selection service is based on the condor middleware. In particular, it makes use of the Condor match making service [149] to select grid resources [150].

### 3.4.1 The Problem of Job and Data Co-location

The literature presents studies on different scheduling algorithms that address the problem of job and data co-location. Adaptation to the dynamism of the grid is the common denominator of four works that we discuss hereby. Alhusaini, Prasanna, and Raghavendra [152] study different adaptive algorithms to select computing, data, and network resources. They assume complete central knowledge of the internal structure of the job dependen-

cies, expressed in the form of a Directed Acyclic Graphs (DAG) (sec. 3.2.4). The scheduler is also assumed to have enough information to estimate the time to completion of each task. Different heuristics are applied, such as either submitting the shorter task or the longer task first. These heuristics are applied to a level-by-level or a greedy approach to job scheduling. In a level-by-level approach, independent tasks of the DAG that are at the same depth are compared then scheduled. In a greedy algorithm, tasks that are deeper than the others, but ready to run, are also considered for scheduling. The algorithms use resource reservation to achieve scheduling to a collective set of job and data resources. It is shown that this collective scheduling improved the time to completion of the jobs of 30% with respect to individual resource selection. Shi, Jin, Qiang, and Zou [153] use also resource reservation, introducing algorithms based on job completion deadlines. The algorithm compares the estimated time to completion of a set of jobs ready for scheduling. The estimate includes data movement and running time. The algorithm submits first the job estimated to end closest to the deadline, if the deadline can be met. If the deadline cannot be met, it uses resource reservation for network and computing resources before submitting the job. The algorithm has been tested on the campus grid of Huazhong University of Science and Technology, Wuhan, China, on 5 clusters with less than 16 nodes. The algorithm was compared with a greedy algorithm for the number of missed deadlines and it performs up to 50% better. Park and Kim [154] describe Chameleon, a system for job scheduling built on top of Globus and based on a cost model. The model analyzed defines cost in terms of time to completion of the job, similarly to the two papers discussed above. Chameleon assumes full knowledge of the computing resources and use the Network Weather service for information on the network conditions. When scheduling a job, the system analyzes five different running scenarios, depending on the locality of the data with respect to the computing resources. The scenario with the lowest cost determines the scheduling strategy. The performance of the system has been tested on a grid of 9 sites with two different applications. Casanova, Obertelli, Berman, and Wolski [155] present AppLeS, a system capable of interfacing to different grids middlewares. Ap-

pLeS specializes in the scheduling of parameter sweep applications, a category of computations that include montecarlo generation. When jobs are scheduled, the system places files strategically for maximum reuse.

Ranganathan and Foster discuss in [156] and later in [76] the effects of asynchronous data replication on the scheduling problem. Unlike the four works discussed above, they assume a complete distributed model. On the other hand, each job is scheduled independently from the characteristics of the others. The model is based on two main components: a global job scheduler and a data scheduler. The global job scheduler uses four algorithms to submit jobs: randomly, always to the local site, to the least loaded site, and to the site with most of the data required by the job present. The data scheduler keeps track of the popularity distribution of the data and replicates popular data to other sites. The algorithms investigated for data scheduling are three: do not replicate data, asynchronously replicate data to a random site or to the least loaded site. The authors use a simulator to compare the twelve combinations of algorithms against three different metrics: response time, cpu idle time, and bandwidth utilization. The study shows that asynchronous data replication to the least loaded site typically increases the efficiency on the metrics only of a few percentage: the two strategies seem to be equivalent. Sending the jobs where the data are substantially improves the figures on the metrics only if asynchronous data replication is used. If the data is not replicated, in fact, the site with the most popular dataset becomes busy processing jobs and its performance degrades. Their conclusion is that coupling data and job scheduling is not necessary when using this model.

A completely different approach to job and data co-location consists in defining execution domains, where resources are bound together according to different affinity criteria. Unlike the works on adaptive scheduling algorithms, the two described hereby do not rely on a scheduler to select from a global pool of computing and data resources, as the execution domains define the possible resource associations. Both papers rely on classad and the condor system to implement their solution. Thain, Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and Livny [157] define execution domains as

"communities", where computation resources advertise the characteristics of the storage to which they are associated. Jobs express their requirements on storage, thus selecting computational sites. It should be noted that this mechanism does not promise the storage to the job and that the storage cannot impose any requirements on the jobs. The match making, in fact, is carried between the job and the computing resources. The system was used to run montecarlo generation for the CMS experiment. It is shown that creating communities improves CPU consumption and completion time over independent resource selection. Basney, Livny, and Mazzanti [158] extend the same concept developing a framework that defines "execution domains". Agents, or domain managers, automatically place replicas, dynamically extend the domains, and ensure that applications run within the domains. Replica placement policies are not discussed in the paper.

The classad technology is also at the basis of two studies related to job and data co-location. Vazhkudai, Tuecke, and Foster [159] implement a storage broker that uses match making between job and storage devices. In this model, a replica catalog is first queried to find the location of the required data, then match making selects the best storage from the device list. The paper does not explore completely the interplay of storage selection with computing resource selection, a possible topic of interest for further investigations. Raman, Livny, and Solomon [160] describe an extension to the classad technology, called gangmatching, that allows the evaluation of multi-party policy expressions by multilateral match making. Unlike the technique described above for [157], gangmatching allows multiple entities to be exclusively promised to each other. The paper does not concentrate on the problem of job / data co-scheduling, even if the technology could be used to solve it.

Li, Groep, Templon, and Wolters [161] implement a solution to the load balancing problem for LCG. The LCG resource broker queries each site for an evaluation of how long the job would stay idle in the queue of the local scheduler. A server running at each site uses a statistical model to produce this estimate, using a simulator of the local scheduler and the status and history of the job queue. This mechanism is similar to the SAM-Grid match

making call-out to the SAM stations to determine the percentage of files cached at the site for a given job (sec. 3.4.3).

The SAM-Grid provides a prototypical solution for the problem of job and data co-location. This is achieved by configuring the resource selector to call-out to the site-deployed SAM stations at the time of job/resource match and by using SAM to pre-stage data as soon as the job enters the site. This mechanism, described in section 3.4.3, can be improved in different ways. First it is adaptive only with respect to the condition of the data handling services. The conditions of the network bandwidth and of the job handling services at the resource are not included. Second, resource utilization policies are not currently part of the algorithm. Third, the algorithm minimizes data transfers on a per-job basis, but does not attempt any global optimization on the metrics relevant to the virtual organization, such as time to completion, bandwidth utilization, etc. In other words, jobs are matched to resources without considering the description of other jobs. These refinements to the job/data co-location algorithm implemented by the SAM-Grid can be addressed in the future if necessary.

### 3.4.2 The Information Service

The SAM-Grid architecture relies on a central information service, which is used for resource selection as well as a component of the naming and monitoring services. Resources and job-related services register with an *information collector*, exposing characteristics of their interfaces and internal statuses. For example, submission site services advertise their current address, so that other entities can manage jobs or acquire details on the job queue. Execution sites, instead, advertise attributes such as their grid entry point, the URL of the local XML database, the name of the local SAM data handling services, the address of the SAM naming service, and other aggregate information on the cluster (see also chap. 2). It is worth noting that while the technology used to describe resources and services is the Condor Classad [151], most of the syntax and the semantics of the description are specific to the SAM-Grid. In order to adapt to the dynamism of a grid

system, the collector treats the incoming information as a soft-registration. This means that the registration is automatically discarded after a configurable amount of time, thus reducing the impact of stale information. It should be noted that this does not prevent stale information from being in the system. In fact, this is one of the reasons why the resource selection should be considered as a "recommendation" only.

The collector acts effectively as an information system, populated by resources and services advertising their characteristics. For the SAM-Grid, such advertisement system presented the following problem. The Condor framework and, therefore, the job management component of the SAM-Grid are integrated with the Classad language. A classad is a "flat" list of attribute/expression pairs [2] (figure 3.9 shows an example). On the other hand, the SAM-Grid site configuration description uses a hierarchical representation of the resources (sec. 2.1.6). Resources and services at a site, in fact, are represented in XML and stored in the XML database at the site (figure 2.6). The advertisement framework, therefore, had to provide a mechanism to decompose the site configuration hierarchy in a set of flat descriptions, represented in the form of Classads.

In order to do this, various algorithms can be applied. The site hierarchy is composed of nested services and resources. To run a job, a set of particularly important services and resources is always necessary, but, in our case, never more than one at the time. For example, a job runs on one cluster, accessed by one gatekeeper and one job-manager, and the job's data are handled by one sam station. The algorithm that we have implemented to achieve the classad generation looks up for key resource and service attributes. Each classad must contain a full set of the key attributes. The site hierarchy is traversed depth-first, looking for the attributes in the expected order of nesting. If a branch does not contain the full set, it is pruned and will not generate any classad. For those branches that generate classads, additional attributes in the hierarchy are also included for completion, such

[2] A newer version of Classad allows the nesting of classads. This enables the expression of hierarchies within the Classad language. However, as of Condor 6.7, the new Classad library is not yet integrated with the Condor framework.

*Idealized site configuration in XML format*

```
<root>
  <key_service_1 attribute='A'>
     <key_service_2 attribute='B'>
         <key_service_3 attribute='C'/>
     </key_service_2>
     <key_service_3 attribute='D'/>
  </key_service_1>
  <optional_service attribute='E'/>
</root>
```

---

*Idealized site configuration in classad format*

```
key_service_1_attribute    = A
key_service_2_attribute    = B
key_service_3_attribute    = C
optional_service_attribute = E
```

Figure 3.8: An idealized example of the conversion algorithm of a site configuration in XML (top) to classad (bottom). The job needs key_service 1, 2, and 3 in order to successfully run. The algorithm traverses the XML tree depth-first finding 2 sets of key_services: 1, 2, 3 AND 1,3. The complete set of key_services and their attributes are transformed in a classad, adding the optional_service at the root level.

as all the children of the document "root". In addition, the advertisement framework can include attributes with dynamic values in the classad, such as the XML database entry point, the data handling system naming service, or the amount of free storage space at the site. Figure 3.8 shows the idea described above with an idealized example. Figure 3.9 shows a real example based on the site configuration of figure 2.6.

In more detail, the advertisement server operates in cycles. At the end of each cycle, a set of classads is sent to the information collector. New cycles are started every few minutes. This time is appropriate, in the case of the SAM-Grid, because the information useful for resource selection typically changes on a longer time scale. Input to the advertisement server is the

XML representation of the site configuration. Each cycle is composed of three phases: pre-processing, processing, and post-processing.

The pre-processing phase can be used to prune the site hierarchy from information irrelevant for resource selection. This phase is, in practice, not used in the SAM-Grid.

The processing phase implements the rules to "flatten" the hierarchical structure. These rules are expressed using an XQuery transformation script. The script decomposes the site configuration tree in the set of all possible branches. A branch is the set of all the nodes from leaf to root. It should be noted that a branch is a linear structure that can be mapped to a "flat" format. This property is used at the end of the cycle to transform the selected branches in classads. Relevant branches are selected applying filters to the set of all the branches. The filter used in the SAM-Grid requires that each key attribute appear once and only once, with a certain relative position in the hierarchy. For example, the typical transformation rule requires that a station is a "descendant" of a gatekeeper and that a gatekeeper is a "descendant" of a cluster (figure 2.6). The nodes of the remaining branches are then decomposed into attribute/value pairs, which is the form used to represent resources in classad format. We utilize the rule that decomposes nodes of the form

```
<tag attr1=valA attr2=valB>
```

into XML nodes of the form

```
<attr name="tag_attr1_" value="valA">
<attr name="tag_attr2_" value="valB">
```

Since, ultimately, classad attributes must have a unique name, nodes of the same branch that would be transformed into attributes with the same name must be treated differently. For example, following the transformation rule above, two nodes of the form

```
<tag attr1=valA>
<tag attr1=valB>
```

would generate two attributes with the same name: tag_attr1_ . In these cases, we decided to break the homonymy by generating attribute names of the form tag_attr1_valA_ and tag_attr1_valA_ . The corresponding attribute value required by the syntax of the classad format is, in reality, not meaningful and is set to the default string "Present". After the node transformation, each branch is a linear structure made of nodes in attribute/value pairs format. Each branch contains static information about a set of site resources needed by a job. In conclusion, at the end of the processing phase, the branch is transformed into an XML representation of a classad.

The post-processing phase of the advertisement cycle adds attributes with dynamic values to the classad. A set of sensors gather information such as the amount of space available in the temporary local data storage ("durable location", sec. 3.2), the number of grid jobs currently running at the site, etc. Figure 3.9 shows a real example of a classad generated from the site description of figure 2.6.

### 3.4.3 The Match Making Service

So far we have talked about the SAM-Grid information service, or information collector, and the system that populates it, the advertisement framework. These are two fundamental components to implement abstract to concrete job description and, more in general, resource selection. In the condor framework and in the SAM-Grid, a third component, called negotiator, is responsible for comparing this information with the characteristics of the jobs in order to find a job / resource match. Within this framework, the job description can refer to resource attributes using their names, as specified in the resource classads. These attributes can be used in expressions that define the behavior of various services, such as resource selection, job environment preparation, job submission service, etc.

Using this mechanism, a job can define its attributes in terms of *abstract* resource characteristics, or attribute names, letting the framework dereference these names in *concrete* values for the job attributes (chap. 3.4).

The resource selection is processed in match making cycles and it is

```
MyType                "Machine"
TargetType            "Job"
Requirements          TRUE
gatekeeper_url_       "apex.cs.wisc.edu:2119/jobmanager-samgrid"
sam_nameservice_      "IOR:000000000000002a49444c3..."
Name                  "d0ppdg.d0.prd.apex.cs.wisc.edu:2119/jobmanager-samgrid"
DbURL                 "http://apex.cs.wisc.edu:7080/Xindice"
station_name_         "d0ppdg"
station_experiment_   "d0"
station_universe_     "prd"
jobmanager_name_      "jobmanager-samgrid"
gatekeeper_location_  "apex.cs.wisc.edu:2119"
cluster_architecture_ "Linux+2.4"
cluster_name_         "D0PPDG-Cluster"
schema_version_       "1_1"
site_name_            "Wisconsin"
local_storage_path_   "/sam/disk"
local_storage_node_   "apex.cs.wisc.edu"
StartdIpAddr          "<198.51.254.108:36359>"
LastHeardFrom         1116988844
```

Figure 3.9: One of the two classads generated by the advertisement framework from the site description of figure 2.6. The other classad (not shown) only changes in the attributes *Name*, *gatekeeper_url_*, and *jobmanager_name_*, which all use the value of the job manager *jobmanager-runjob* instead. The attributes *sam_nameservice_* and *DbURL* are dynamically generated and do not show up in the static site description of figure 2.6. The attributes *MyType*, *TargetType*, *Requirements*, *StartdIpAddr*, and *LastHeardFrom* are required keywords used internally by the Condor framework. The algorithm used to generate this classad was sensitive to the hierarchy of resource/service attributes *cluster*, *gatekeeper*, and *station*.

configured in the job description, by the use of *requirements* and *rank* attributes. Periodically, every submission site is queried about the details of the jobs queued therein. For every job, resources are initially skimmed by looking at the job requirements. The requirements express constraints on the value of the resource attributes. The remaining matching candidates are then ranked according to various configurable algorithms. When the "best" match is selected, the job description is enriched with information from the resource.

Figure 3.10 shows an example of a condor job description file used by the SAM-Grid. The condor JDF is translated by the framework into a classad. The classad is used for the actual match making, while the condor JDF expresses the same content in an easily readable format. The condor JDF is also relevant in this context because it is the result of the translation from the user-defined SAM-Grid JDF (sec. 3.2). The syntax TARGET.attribute_name allows the job description to refer to the name of the resource attributes (see figure 3.9); $(attribute_name) is substituted with the value of the corresponding named attribute at any resource matching attempt; $$(attribute_name) is substituted with the value of the named attributes *after* the match has been found; +attribute_name is copied verbatim from the condor JDF to the job classad.

It is worth commenting a few characteristics of the example of figure 3.10. The value of the attribute *globusscheduler* is used by the submission site to submit the job to the selected resource. In the SAM-Grid, the value of the attribute is made *concrete* after the resource has been selected and it is defined by the attribute *gatekeeper_url_*, as advertised by the resource. The *globusscheduler* attribute is now a standard of the Condor middleware and was one of the modifications introduced to the Condor match making service as part of the collaboration of the Condor team with the SAM-Grid [150]. Other attributes are also made concrete after the resource has been selected. In particular *DbURL* points to the entry point of the site monitoring XML database. This is used by the user monitoring interface to locate detailed information about the job, in case the information in the classad is too high-level. Other attributes, such as *station_name_* and the *name* of the resource

classad, are used to set the job environment variables, using the *environment* job attribute. Some of the attributes of in the figure are used to program the behavior of the submission service (sec. 3.3). In particular, they control the rematch / resubmit policies, disabled in the example, and the clean up policies after job submission (*periodic_remove*). Finally, the attribute *arguments* is used to program the behavior of the Grid/Fabric interface 4.

In this example, the resource is univocally selected by the *requirements* attribute, as the name of the sam station, a binding condition, univocally identifies a resource. In general, the system would also need to rank the set of resources that satisfy the requirements. This is achieved by the use of the *rank* attribute.

When we started designing the system, the rank could be a generic expression of attributes from both job and resource classads. At the time, we were working with a prototypical implementation of analysis jobs and we wanted to send the jobs to the resource that had already in its storage elements most of the data requested by the job. A simple expression to define the rank was not enough to solve our problem. The amount of data at every site, in fact, may easily reach tens of thousands of files, making thus impractical sending the information to the match making service for every execution site, and, in particular, sending it via a Classad. It is recommendable to limit the amount of information contained in the registration Classad to a few dozen attributes. The use of this technology for much larger data volumes, in fact, has not been fully investigated, but it is likely to lead to an inefficient match making process.

To overcome this problem, the resource selector of Condor has been modified to call externally provided functions when evaluating a match. Both job and resource descriptions can define any attribute of their Classad using these functions, passing to them as arguments any other attributes of the two Classads. The implementation of the function is provided to the negotiator in the form of a shared object. A configuration file maps the name of the function, as specified in the classad, to the function symbol of the shared object. Using this mechanism, the SAM-Grid team has developed a function that queries the remote SAM stations at the time of matching. The

```
universe = globus
+JobType = "dzero_monte_carlo"
+station_universe = "prd"
+station_experiment = "d0"
+jobmanager_name = "jobmanager-samgrid"
+DbURL = "$$(DbURL)"
+RequestId = "20214"
+NumEvents = 12750
globusscheduler = $$(gatekeeper_url_)
executable = /tmp/garzogli_samadams.fnal.gov_222357_20315.tar.gz
requirements = TARGET.station_name_ == "samgfarm" &&
               TARGET.jobmanager_name_ == $(jobmanager_name) &&
               TARGET.station_experiment_ == $(station_experiment) &&
               TARGET.station_universe_ == $(station_universe)
environment = MATCH_RESOURCE_NAME=$$(name); SAMG_JID=$(ProjectIdVal);
              SAM_STATION=$$(station_name_);SAM_QUALIFIER=prd;
              SAMG_JOB_TYPE=dzero_monte_carlo;SAM_USER_NAME=garzogli
match_list_length = 1
globus_rematch = false
globus_resubmit = false
periodic_release = false
leave_in_queue = true
periodic_remove = ( (CurrentTime - EnteredCurrentStatus) > 1209600 &&
                    JobStatus == 4 )
ProjectIdVal = garzogli_samadams.fnal.gov_222357_20315
+ProjectId = "$(ProjectIdVal)"
arguments = \"dzero_monte_carlo\" \"--requestId=20214\"
            \"--gridId=$(ProjectIdVal)\" \"-v\"
            \"--numEvents=$(NumEvents)\" \"--d0ReleaseVersion=p14.07.00\"
            \"--jobFileDataset=hynek_jes_group_15400 --jobFileDatasetId=228115\"
            \"--phaseDataset=CKKW_Rand_1.W+4run4.1.io --phaseDatasetId=238056\"
            \"--numEventsToSkip=12250\"
            \"--minBiasDataset=overlapset_mcp14_cteq5l-tuneA_simulated
              --minBiasDatasetId=223422\" \"--eventsPerFile=250\"
queue
```

Figure 3.10: The condor job description file (JDF) (shortened) resulting from the SAM-Grid JDF of figure 3.4

query returns the percentage of overlap between the job dataset, available in the job classad, and the files cached at the remote station, available in the resource classad. Therefore, the algorithm adapts to the conditions of the grid with respect to the availability of the data handling services and the placement of the replicas. Internal information caching was also implemented to improve the matching time.

# Chapter 4

# The Execution Site

The mandate of the SAM-Grid project was making available widely distributed computational resources to the DZero and CDF communities. The SAM-Grid was designed assuming that the participating resources where shared and distributively owned. Since the resources were shared, the deployment of the software had to be limited to a few machines. That is, no software could be run at the worker nodes of a participating cluster. Since the resources were distributively owned, the grid could not impose special configurations of site services, such as the local scheduler or the intra-cluster data transport mechanisms, in order to make them accessible through the grid. In other words, when designing the SAM-Grid, we had to ask ourselves how to run on non-dedicated resources and how to foster site autonomy.

## 4.1 Shortcomings of the Standard Middleware

Working on the development of the SAM-Grid, we soon realized that the "standard" grid technologies were not flexible enough, out of the box, to satisfy these requirements. Even in the simple case of interfacing grid jobs to the local batch system, the grid software supported only very standard configurations, which, in practice, we did not find in the three pilot sites of our deployment. Understandably, the administrators were not willing to change the configurations of their batch system. First, the configuration reflected

the policies of the local site. Second, each cluster had already a user-base used to the current site configuration. Third, wasn't the grid supposed to respect site autonomy? The adaptation of the grid to the configuration of the local batch system became a priority for our project.

It turned out that adaptation to the local scheduler was only a task within a complex project. Central to the SAM-Grid architecture was SAM, the data handling system of the experiments. SAM is a mature infrastructure and the only accepted way for a physicist to access the experiment data. DZero users commonly notified SAM when submitting data intensive jobs to their local batch systems, in order to initiate data pre-staging while the job was idle in the scheduler queue. It was expected that the SAM-Grid could support data pre-staging on the grid as well. But when should SAM be notified of a grid job entering the system?

- The time of job submission was clearly too early. The job was not yet matched with a site, so it was impossible to know at what site to start the data pre-staging.

- The time of job/resource matching was again arguably too early. If stale information was in the system, in fact, the match maker could initiate data pre-staging at a site that, in the end, was possibly not accessible by the job management system. This would translate into a waste of storage resources, given that SAM temporarily reserves a portion of such resources for the data.

- The time of the job entering the site, instead, seemed the appropriate approach: not only this was the same strategy that was used when submitting jobs locally, but also this "lazy" approach overcame the problems of the two methods above.

We soon realized, however, that the "standard" grid software could not notify any local services, other than the local batch system, that a job entered the site. More generally, the standard interface between the grid and the local services and resources, commonly called the *fabric*, was not comprehensive enough. This was potentially a problem when running complex

jobs, such as data-intensive jobs. In other words, we had to ask ourselves how to coordinate fabric services to execute complex jobs.

During our research we observed another shortcoming of the standard middleware. The standard middleware focuses mainly on the management of resources. It provides interfaces to local job schedulers, to data storage systems, to network management systems, to information catalogues, etc. On the other hand, complex applications often rely on an application management system to assist with the preparation of the job environment, the execution of the job, and the interaction with the resources. The standard middleware lacked a set of interfaces to interact with application management systems.

High energy physics applications commonly use application management systems, such as Runjob [119]. Typically, an application management system provides facilities to manage the workflow of the application, running tasks in the appropriate order and managing their input/output, so that the output of a task is made available as the input of the appropriate subsequent tasks. An application management system participates in the selection of resources, interfacing to the resource management system and considering the characteristic of the running application. For example, it can recommend different local scheduler queues, which generally express local running policies, depending on the type of application. As another example, it can determine the ideal multiplicity of concurrent processes for running a job: an application that processes a dataset of $N$ files has typically a maximum multiplicity of $N$ processes, while an application that produces $M$ montecarlo events has a multiplicity that depends on how long it takes to produce an event, how many computational resources are available, the maximum time a process can run on a resource, the subsequent processing steps for the output, etc. Yet another example is the choice of a data access queue depending on the type of application. Let's suppose that there are $N$ jobs running a certain application on a cluster and that the application requires multiple files before running. This is typical of merging applications, whose function is concatenating small size files in larger ones (sec. 3.2.4). Let's also suppose that there are $100 \times N$ jobs running a different application that re-

quires one file to run. It is often not efficient to let both types of applications access a single data queue to retrieve the files. If the data access requests are interleaved, it can take a long time for the merging jobs to start. Organizing the access requests on multiple queues depending on the application can be more efficient. For a discussion of these examples, the reader is referenced to section 3.2. All of these choices can be determined by the application management system, which knows the details of the job, and not by the resource management system. Because of the lack of interfaces between the two layers, the SAM-Grid could not use directly the standard middleware and had to integrate some application-specific knowledge of the common high energy physics applications with its resource management components.

Using the standard middleware, we also realized that the load to the gateway machine imposed a scalability limit on the number of jobs running at the cluster. The Globus Gatekeeper forks a process for every job entering the cluster. This process, called the job manager, is the contact point for the grid to manage the job. This limits the number of jobs that the cluster can run to a couple of hundreds, which is the typical limit for the average commodity computer. Already in 2001, it was pretty common finding clusters with hundreds of machines. In addition, since the Grid was a new concept and its services not very exploited, the sites generally made available a few cheap low-performance machines to act as gateways. In this scenario, we clearly had a problem of scalability with the implementation of the standard middleware. The SAM-Grid addressed the problem by rewriting the job manager, in order to submit multiple batch processes for a single grid job entering the Gatekeeper. The statuses of the batch processes were aggregated and returned to the grid as the status of the grid job. A single job manager became the contact point for multiple related jobs. Other groups, such as Condor, use the gatekeeper itself to kill and selectively restart the job managers, in order to reduce the number of processes continuously running at the gateway. We don't like this solution because it achieves scalability by altering the normal behavior of the standard middleware in an intrusive way (killing components of the architecture), instead of finding a solution that adhere to the accepted architectural standards. We believe that local job

multiplicity management as developed by the SAM-Grid is a more general solution.

More generally, we experienced that executing complex jobs, such as data intensive jobs, required the ability to control the load of all the resources participating to the computational activity. In addition to the aggregation of batch processes at the gateway machine, the SAM-Grid implemented queues for data access to the local storages, in order to control the load of the data server machines. Using job aggregation, it was possible to pool expensive database accesses to information common to all the aggregated jobs. The standard middleware did not offer any support to address these broader scalability concerns.

Another problem of the standard middleware was the lack of robustness in case of failures of fabric services. In particular, when interacting with a local scheduler, we observed that the standard middleware could not properly react to errors raised by the scheduler or to conditions of network time out. We had to ask ourselves how to overcome temporary failures of fabric services. The SAM-Grid addressed this issue by implementing the throttling of the flow of grid jobs to the resources, in order to limit the load to the services, by caching information such as the statuses of the jobs, and by introducing retrials in the interaction with fabric services, including the local scheduler.

In conclusion, we had to build an infrastructure that allowed running of jobs on non-dedicated resources, fostering at the same time site autonomy. The standard middleware lacked in comprehensiveness in interfacing to fabric services, since the local scheduler was the only service integrated. It also lacked flexibility in the adaptation to the batch system configurations and it lacked robustness in the interactions with the fabric. It presented various problems of scalability and limitations in its interfaces, especially in the interactions with application management systems.

To overcome these problems, we reimplemented the interface between the grid and the fabric within the framework of the Globus Toolkit. In the following sections, we describe how this interface coordinates the interactions with multiple fabric services, including the local scheduler (sec. 4.2),

how it manages the job environment preparation (sec. 4.3) and how it integrates aspects of application management, including process splitting and aggregation (sec. 4.3.2.2).

## 4.2 Local Batch System Adaptation

The SAM-Grid was designed to execute complex high energy physics applications on a grid of resources. Such applications assume the availability of several services at the fabric, including batch scheduling, data handling, environment preparation, scratch management, and monitoring. The standard middleware provides implementations of grid-to-fabric interfaces that interact with the local batch systems only. The reference implementations of the standard middleware were not comprehensive enough for the use cases that the SAM-Grid was addressing.

The SAM-Grid team used the framework of the Globus Toolkit to implement a broader grid-to-fabric interface [162, 163, 164]. When a job enters the site, the SAM-Grid interface notifies the local data handling system to initiate data pre-staging (sec. 3.4.3), it interacts with the job environment preparation service (sec. 4.3) and the configuration manager (sec. 2.1), it notifies the monitoring system of an incoming job (sec. 2.2.3), and it gathers information from the application in order to compute the appropriate multiplicity of the batch processes. As a last step, it interfaces to the local scheduler.

Because of the variety of batch systems available, it was necessary to abstract the interaction with the batch system, in order to modularize the implementation. This layer of abstraction was called the "batch system adapter". For a given site, the batch system adapter allows the definition of multiple adapters, each defining a set of interfaces, or commands, to the underlying scheduler. Using the standard middleware as a reference, we initially defined three basic interfaces to a batch system: "submit job", "lookup job status", "cancel job". The interfaces were later extended to implement commands such as "collect output" and to gather general status information, such as "display policies" and "display system load". The interfaces

could also be configured to interpret the output of the batch system commands. This is used to parse relevant information, including the local job id after submission, the status of the job after lookup or the error messages after the invocation of any commands. Figure 4.1 shows the batch adapter configuration of a site as an example.

For our implementation of the basic interfaces to the batch system, we initially thought of reusing the code provided by the standard middleware. Yet, after our first experience deploying the SAM-Grid, we concluded that they were not flexible enough to include most of the resources of DZero and CDF. Even at sites running the same batch system, we observed that different administrators frequently configured the batch system differently because of local constraints, thus requiring the local users to submit jobs using slightly different commands. In some other cases, the terms of the agreement to use the resources could be respected only by adding special attributes to the job submission request. For example, when DZero submits jobs to the condor cluster of the University of Wisconsin at Madison, the job description file must include special attributes to prevent job eviction. Another example is the CCIN2P3 computing facility in Lyon, France. Jobs that access data from the HPSS mass storage system can be submitted with a special flag. In case of downtime of HPSS, the batch system is programmed to hold the jobs until the storage is back on line. These attributes and submission options are all nonstandard and site specific. Other grids, such as LCG [37], solve this problem by exposing to the grid information service the interface of the local scheduler. After the resource is selected and the grid job is scheduled, the local scheduler interface is passed to the grid-to-fabric interface of the remote resource, so that the job can be properly locally submitted. Incidentally, this mechanism is deemed to be not flexible enough to encompass all possible configurations of local schedulers. It is believed, in fact, that "modern batch systems are too complex and dynamic to summarize their behavior in a few values in the Information Service" [165]. In any case, we concluded that we could not reuse the implementation of the Globus Toolkit out of the box.

In addition to the lack of flexibility, the standard middleware was lacking

```
Station: fnal-farm
  Default Adapter: grid
  Available Adapters: ['grid']
    Adapter: grid
      Available Commands: ['job submit command', 'job lookup command',
                           'job kill command', ]
        Command: /bin/sh -c '. /local/products/ups/etc/setups.sh;setup fbsng;\
                ${SAM_BATCH_ADAPTER_HANDLER_DIR}/sam_fbsng_handler.py job_submit \
                --project=%__USER_PROJECT__ --executable=%__USER_SCRIPT__ \
                --arguments=%__USER_SCRIPT_ARGS__ \
                --stdout=%__USER_JOB_OUTPUT__ --stderr=%__USER_JOB_ERROR__ \
                --queue=SAMGrid --jobtype=%__USER_APPLICATION_TYPE__'
          Type: job submit command
          Known Outcomes:
            Exit Status: 0
            Outcome Description: Success
            Exit Status: 0
            Expected Output: %__BATCH_JOB_ID__
            Exit Status: 1
            Outcome Description: Failure

        Command: /bin/sh -c '. /local/products/ups/etc/setups.sh;setup fbsng; \
                 ${SAM_BATCH_ADAPTER_HANDLER_DIR}/sam_fbsng_handler.py job_lookup \
                 --project=%__USER_PROJECT__ --localJobId=%__BATCH_JOB_ID__'
          Type: job lookup command
          Known Outcomes:
            Exit Status: 0
            Outcome Description: Success
            Exit Status: 0
            Expected Output: JobId=%__BATCH_JOB_ID__ Status=%__BATCH_JOB_STATUS__
            Exit Status: 1
            Outcome Description: Failure

        ...
```

Figure 4.1: The configuration of the "grid" batch adapter for the "fnal-farm" SAM station (shortened). The adapter defines three interfaces with the batch system: job submit, lookup, and kill/cancel (not shown). The commands are implemented via the sam_fbsng_handler.py idealizer. The command arguments are defined using templates (e.g. %__USER_APPLICATION_TYPE__), which are substituted by the batch adapter client. Templates are also used by the client know how to parse the command output.

in robustness. In our experience, all different types of local schedulers in our grid failed at some point to return correctly from a client command. Because the grid polls periodically the status of the jobs, job polling was the command that showed most of the problems. The typical error condition was a timeout, but we also saw rare occurrences of wrong results, for example when the BQS batch system [166, 167] could not find jobs just submitted for a minute. In some other cases of particular high load, the batch system could not sustain the frequency of status polling, returning a "denial of service" error. Another interesting failure mode was what we called the "black hole" effect. Misconfigured worker nodes cause the jobs running there to fail very rapidly. Nevertheless, the local scheduler keeps sending idle jobs to that worker node, until all the idle jobs have failed (sec. 4.2.1).

Apart from the black hole effect, the other problems mentioned are transient and of no consequence in case of human interaction. For example, a user would simply re-issue a command in case of timeout. The standard implementation of the grid, instead, immediately considered the job failed. In general, we needed an implementation that would *idealize* the interaction with the batch system, overcoming temporary failures of the service. This was achieved by implementing a batch system handler, called batch system idealizer, that would increase the robustness of the batch system as seen by the grid.

In our implementation of the idealizer, the interactions with the batch system are retried with a random exponential back-off strategy. Typically the first retrial is after a few seconds, then after a few dozens, and so on. Taking this approach, we assume that the failure is temporary, that it is a denial of service, and that it is caused by peak of activity at the server. In practice, these assumptions are satisfied most of the time. In case of status polling, the idealizer also caches the job statuses. Since the grid typically polls the job status every few minutes, this gives us an opportunity of returning a stale information and restarting the exponential retrials at a new polling cycle. Analyzing data for a period of 4 months (June to September 2005), we have estimated that the probability of a job failing because of a temporary failure in the communication with the batch system

is lower than $7 \times 10^{-6}$. Statistically, that corresponds to a failure rate more rare than a job failure every 150,000 jobs.

In summary, the standard grid-to-fabric interface did not interact with services such as data handling, monitoring, configuration management, etc. We have reimplemented the grid-to-fabric interface to overcome this lack of comprehensiveness. Our implementation interacts with the batch system via a layer of abstraction called "batch system adapters". The adapters define the interface with the underlying batch system. The implementation of these interfaces is flexible, in the sense that they can be programmed to include site-specific characteristics of the batch system client commands. In addition, the implementation idealizes the batch system, in the sense that it enforces a robust interaction with it. In conclusion, the SAM-Grid grid-to-fabric interface implements a comprehensive interaction with fabric services and achieves flexibility and robustness in the interaction with the local batch system.

### 4.2.1 The Black Hole Effect

In order to increase job throughput and efficiency at a local computing resource, we have investigated mechanisms to determine and react to failures caused by the mismatch between the application requirements and the execution host [168]. In particular we deal with failures that occur when the execution host fails to provide a required service or a set of services to the application, resulting in the application crashing. In this context, a service is a library or a software product and it should not be confused with the term "grid service", as used throughout the rest of the dissertation.

Applications, typically, rely on a set of basic services offered by the execution environment, such as compression utilities and system libraries, and usually do not explicitly specify these in the job description to the resource management system. Compute nodes in clusters are heterogeneous in terms of the underlying architecture or the software environment and utilities. Service version conflicts may result in the application failing, typically quickly, on a particular compute node, which can potentially have a cascading ef-

fect. The entire set of queued batch jobs, running the same application, gets scheduled to the compute node that fails the application, resulting in low throughput and efficiency. This phenomenon is what we call the Black Hole Effect. The compute nodes which malfunction with respect to the application are called Black Holes.

A characteristic of the black hole effect is, therefore, a high turnaround on a particular node. The grid software should be able to determine whether the jobs are consistently failing on that node, or the node is just a combination of powerful processing element and large memory. Black hole detection is aided by the application specifying a minimum duration for the job to execute ($d_{min}$). From empirical data, $d_{min}$ can be estimated within an error margin. $d_{min}$ is expressed in terms of wall clock time for a given architecture and a given operating system. For example to simulate 250 physics events of a DZero Monte Carlo simulation job, it takes at a minimum 8 hours ($d_{min}$) between the time it is scheduled by the batch system and the time that the error and output logs are returned. The reference architecture is an Intel x86 with 512 mega bytes of physical memory, 1024 mega bytes of virtual memory, 1GHz of clock speed, running a Linux 2.4.x kernel. In determining if a node is really a black hole, we normalize the values of the compute nodes processing power (processor) and its memory with respect to the reference system, thus arriving at a normalized value of $d_{min}$. Empirical evidence suggests that if the amount of time taken by an execute node to complete a job is less than 60% to 80% of the $d_{min}$ value, then we have a potential black hole.

It should be noted that the black hole effect is an application specific effect. A node which is designated as a black hole for one application might be a perfectly fine node for some other application, if that application does not rely on the same services for its execution.

We believe that the grid-to-fabric interface is the appropriate component to detect the presence of black holes. In particular, for the SAM-Grid, black hole detection and reaction can be implemented in the batch system idealizer (sec. 4.2), which is responsible for increasing the robustness of the interaction of the grid with the local job scheduler. Some basic services are

expected of the fabric to detect the existence of a black hole. These services are incorporated in many popular batch queuing systems and include retrieval of standard output and error streams of the finished jobs, as well as API's for explicitly specifying the compute node on which a job should execute.

We have investigated reactive and proactive approaches to mitigate the black hole effect [168].

The proactive approach of determining whether a compute node can be a potential black hole relies on the application describing in a formal way as to what services it expects from a compute node in the batch system. This approach is a pessimistic approach in the sense that it assumes that there will be potential black holes in a cluster and tries to avoid them by isolating these nodes for the application. In principle, these services can be specified via a service description language. The grid software then has the responsibility of sending small probes (test jobs) to all the nodes in the batch system and analyzes the results of these probes. Persistent information regarding the total number of reachable nodes, nodes suitable for executing the application based on the specified criteria (good nodes), and nodes unsuitable for executing the application (potential black holes) is maintained.

There are two main problems with this approach. First, it is highly dependent on the correct formal specification of the services required from the compute nodes. Second, the persistent information regarding the state of the cluster is not very dynamic, since it depends on the frequency of the probing, and might result in wasted resources.

The reactive approach for dealing with the black hole effect is an optimistic approach. We start by assuming that all the compute nodes in the cluster comply with the applications service requirements. This approach is probabilistic and depends greatly on the availability of statistical data regarding the behavior of the application. By analyzing these data, we associate weights to the list of all available nodes: the higher the weight, the more frequent the record of success of the application on the node. Jobs are scheduled only to nodes with a certain record of success, thus avoiding black holes.

The advantages of this approach are that it is dynamic in nature and does not need any formal specification of what services the compute node should provide. On the other hand, in general, determining the cause of failure is difficult and mining of log files may be necessary. In particular, it is not easy for the infrastructure to determine whether the failure was due to the application failing (property of the application) or to one of the services required by the application failing (property of the node).

If should be noted that hybrid reactive/proactive approaches are possible too. These can mitigate some of the problems of the "pure" approaches alone.

## 4.3 Job Environment Preparation

High energy physics applications seldom consist of a single self-contained executable. Typically, users run applications by providing a list of directives to the experiment software framework. The framework consists of a set of executables and libraries that are assumed to be available at runtime. This approach lets the user run complex computations by composing tasks in high-level physics-specific macro directives, written in scripting languages such as TCL. Users also often complement the framework directives by providing special-purpose executables. In addition, because of the data-intensive nature of the computational tasks, applications rely on the presence of data handling services, such as SAM. Also necessary is the access to databases, typically to retrieve conditions of the data taking, such as calibration constants of the detector and beam luminosity (related to the number of events produced per second). In summary, the job environment of the typical high-energy physics application is comprised of the following components:

- **User-provided information**: it consists of user directives to the framework and special libraries and executables to complement the framework computational code. The library and executable files are generally of small size (Kilobytes or Megabytes) and differ for every

job. When running on the grid, this information is often called the "user input sandbox".

- **Experiment code**: it consists of the experiment offline analysis framework. This code has generally a large size: the offline code of the DZero experiment is 400 Megabytes compressed. The code is the same for every job and it is often pre-installed on resources dedicated to the experiment. On the grid, where the resources are shared, code pre-installation is forbidden by most policies, and the experiment keeps the code available on data handling storages to allow dynamic code installation.

- **Middleware**: these are software products used to assist with the application execution. It consists of application management systems, such as Runjob for DZero (sec. 3.2), or client software of various services, such as SAM, databases, or data transport (FTP, fcp [169], SRM [125], etc.). The size of this information ranges from a few to dozens of Megabytes.

- **Service/Resource configuration**: this information is generally used to locate a service, such as a service discovery service, data handling services, etc. The size of this information is generally less than a few Kilobytes. It is presented in the form of environment variables or small files. It should be noted that while the environment information components mentioned so far are either common to all jobs or coming from the user, the information on the configuration is generally local to the computing environment (e.g. local data handling services configuration, intra-cluster transfer mechanisms configuration, batch system configuration, etc.). One of the challenges of preparing the job environment is, in fact, gathering information from different sources (the user, the site, virtual organization repositories, etc.)

- **User credentials**: these are used to establish a security context with the services contacted by the job. The type of credentials depends on the security model. Most services nowadays allow only strongly

authenticated access. Typical credentials consists of a Kerberos ticket or a X509 certificate. In particular, X509 certificates used with the Globus Security Infrastructure are the most popular security technology for the grid.

### 4.3.1 Problems in the Preparation of the Job Environment

When running applications on dedicated resources, most of the environment is pre-installed and available typically via a shared file system on all the nodes of the cluster. When running applications on the grid, however, none of the necessary environment is generally available and it must be established before passing control to the application. The total amount of information in the environment is generally around 500 MB, when compressed. Most grids do not provide any assistance in the preparation of the job environment, leaving this responsibility to the user or the virtual organization. In this scenario, the environment can be either pushed to the worker node, as a user input sandbox, or pulled by the job itself, before control is passed to the application. The push model is not always a viable solution, because of restrictions on the size of the user input sandbox [1]. The pull model, on the other hand, requires that the user knows how to write code that finds and transports the environment by interacting directly with grid services, such as discovery services, replica catalogues, storage and data transfer services, etc. To simplify the task for our users, the SAM-Grid opted to assist them with environment preparation. The environment preparation service acts transparently and receives input from the job description. In the following subsections, we describe the main problems encountered in the design of the environment preparation service. In section 4.3.2 and relative subsections, we describe the solutions implemented for the SAM-Grid.

#### 4.3.1.1 Information Transport

Different components of the environment are available at different storages either at the site (e.g. local configuration) or external to the site (e.g. the

[1] The maximum size of the user input sandbox on LCG is 10 Megabytes.

experiment framework and middleware). When running on the grid, typically the job or a job wrapper is responsible for retrieving the necessary information and dynamically establish the environment at the worker node. In order to access the source of the information, the job relies on data transport clients. The availability of these clients, though, cannot be taken for granted for the following reasons.

1. Every cluster is natively configured with a different transport mechanism (rcp, kerberos rcp, scp, gridftp, etc.), if such mechanism is available at all. In general there is no agreement among sites to pre-install a common set of data transport clients. Even if, in principle, coding an environment retrieval program would be possible, this heterogeneity would make it difficult to provide a common solution for all the clusters on the grid.

2. It is reasonable to assume that some common data transport client is available close to the worker node e.g. in the form of a compressed archive. VO software, in fact, can be deployed at special machines at the sites e.g. at the gateway node. Nevertheless, it is not immediate making the code available at the worker node. Most batch systems can be configured to transport files to the worker node together with the executables. This capability is called file stage-in. If file stage-in were configured at every cluster, one could stage-in the common data transport client. Yet again, there is no agreement on how batch systems should be configured and, in practice, we cannot rely on the presence of file stage-in capabilities.

3. In general we cannot assume the availability of a shared file system between the worker node and the repository of the data transport client code.

In summary, in order to access the components of the environment, we cannot rely on the native data transport mechanisms of each cluster, nor on the stage-in capabilities of the batch systems, nor on the presence of a

shared file system. The solution provided by the SAM-Grid is based on a self-extracting bootstrapping technique and it is described in section 4.3.2.1.

#### 4.3.1.2 Load Control

Controlling the load of key machines, such as the gateway, is indispensable to achieving production quality. The grid-to-fabric interface accesses services, such as the environment preparation service, that are CPU intensive. Another cause of load to key machines is due to data transfers initiated by the batch jobs. When scheduling grid jobs to a cluster, it is a common occurrence that multiple batch processes start at worker nodes approximately at the same time. In the case of the SAM-Grid, typically dozens of jobs start concurrently, because the grid-to-fabric interface splits grid jobs into multiple (hundreds, in fact) batch processes. Even after solving the "information transport" problem (sec. 4.3.1.1), we still need to provide a scalable data access solution for the jobs to access the job environment components at the gateway and other storage services. The SAM-Grid solution, discussed in section 4.3.2.2, is based on data access queues.

#### 4.3.1.3 Service Configuration Accessibility

When running at the worker node of a grid site, the job has to be notified of what services are available at the site and how they are configured. Often, this information is made available to the job as environment variables pointing to indexing services, such as service discovery services. Optionally, more detailed information on specific services can also be made available as environment variables. Most resource selection services provide the ability of initializing such variables at the time of the job / resource matching. The values of the variables are advertised to the grid information service as part of the resource description (sec. 3.4).

The SAM-Grid system relies for site and product configuration on a configuration framework based on a network of XML databases (sec. 2.1). The address of the site configuration service is made available to the job as an environment variable that is defined at the time of the job / resource match.

All of these schemes assume that the services necessary to successfully complete the job are registered with these grid configuration / discovery systems. In reality, it turns out that there are services that are fundamental for the job, but that are not registered with any of these configuration systems.

One such service for the SAM-Grid is SAM, the data handling service. The configuration of SAM is maintained at the site in the form of configuration files, packaged in a special configuration product (sam_config). Tools parse and transform this configuration into environment variables that are utilized at runtime by SAM. The configuration includes the address of the SAM CORBA naming service, the address of the SAM database, optionally the name of the local SAM station, the addresses of the local calibration database proxies, etc. The values of these parameters are different for the SAM development, integration, and production systems, for clients and servers, and so on. The problem is then how to make the correct configuration files available at the worker nodes. The SAM-Grid utilizes a tool called the product configuration manager in conjunction with the sandboxing framework. This framework is used to transport all the environment components that are available at the gateway node (user input sandbox, user credentials, middleware software, configuration) to the worker node. More details are given in section 4.3.2.3.

#### 4.3.1.4 Environment Uniformity

High energy physics applications are generally implemented through experiment specific analysis frameworks. The framework consists of executables and libraries programmed around the peculiarities of the detector and its read-out electronics. Special algorithms transform electronic signals from the detector into physical quantities related to particle events. As the understanding of the detector improves, these algorithms are refined and new releases of the analysis frameworks are made available to the collaboration. Not all the collaborators are necessarily interested in processing their analyses with the new algorithms. They may have, in fact, results from old data obtained using old algorithms. Results from new data may not be compa-

rable with the old results, if they were obtained using the new algorithm.

The uniformity of the execution environment among such computations is a crucial requirement to produce meaningful physics results. It turns out that the collaboration may be interested in keeping dozens of release versions available at the same time. When running jobs on dedicated resources, it is the responsibility of the administrator to maintain all such versions pre-installed at the cluster. When running on grid resources, pre-installation of software is generally never an option. Yet most analyses assume the presence of the analysis framework. This makes running on the grid more challenging.

To make the code more portable, the DZero experiment has developed tools that recreate the Run Time Environment (RTE) [171] of the various applications. These tools give the users the ability of packaging their programs with all the necessary software dependencies, thus enabling the execution on rather "hostile" computing environments. In addition, these tools make the user responsible for the uniformity of their application environment. These executables, though, have the drawback of being large (in some cases hundreds of Megabytes compressed). Because of this, it is inconvenient to ship them as part of the user input sandbox. Each sandbox, in fact, needs to be transported and temporarily stored possibly in more than one place throughout the lifetime of the job. This leads to an inefficient use of the storage resources. In other words, we are facing the problem of making this code available at the worker node of the grid computing cluster using a resource effective mechanism. The SAM-Grid solution uses the SAM data handling for the distribution of the analysis framework packaged with RTE techniques and uses the sandboxing framework for the distribution of the user directives. We describe our solution in section 4.3.2.4.

### 4.3.2 Solutions for the Preparation of the Job Environment

The SAM-Grid provides a solution for the "information transport" problem (sec. 4.3.1.1), the "load control" problem (sec. 4.3.1.2), and the "service configuration accessibility" problem (sec. 4.3.1.3) using the SAM-Grid sandboxing framework. All of the components needed to establish the environ-

ment but the analysis framework and part of the middleware, are present at the gateway node of the cluster, once the job has entered the site. The sandboxing tools package all components at the gateway in an archive, or "sandbox", and make it available to the worker with scalable data transfers via pushing or pulling mechanisms. The analysis framework and the remaining components of the middleware are gathered directly from SAM storage elements using scalable mechanisms ("environment uniformity" problem, sec. 4.3.1.4).

Before a grid job is submitted to the grid, user-provided information, user credentials, middleware software, and configuration files are available at different sources around the world. After the job has entered the site, though, they are all available at the gateway node. Their presence at the gateway allows the sandbox framework to solve most of the environment preparation problems. Each of these components is available at the gateway for the following reasons.

- **User-provided information**: it is transported at the gateway node from the user's machine using interfaces provided by the GRAM protocol (sec. 1.1). The Globus Gatekeeper provides a data transport server, called GASS (Globus Access to Secondary Storage) [172]. The submission site services (sec. 3.3) use the GASS server to transport the user input sandbox. The user input sandbox is a file (generally an archive) that can be specified by the user via the SAM-Grid job description language (attributes *input_sandbox* or *input_sandbox_tgz*).

- **User credentials**: they are delegated to the Globus Gatekeeper by the submission site. The job uses the user credentials to authenticate with services at the site, such as SAM and storage elements. After delegation, the user credentials are available at the gateway in form of a file.

- **Middleware**: it can be present at the gateway node as products that are pre-installed by the Virtual Organization (VO). While sites do not allow the installation of products at the worker nodes of the cluster,

they generally provide a few special nodes where VO-specific software can be installed. The gateway node is one such nodes. In the case of the SAM-Grid, the middleware pre-installed at the worker node consists of the data transfer client (gridftp and the SAM-Grid GSI configuration, used in our solution to the "information transport" problem, sec. 4.3.2.1), the data queuing client (fcp, used at the core of our solution for the "load control" problem, sec. 4.3.2.2), the jim configuration client software (used to retrieve product/site configuration), and the SAM client (used to solve the "environment uniformity" problem, sec. 4.3.2.4, as well as to retrieve data and additional middleware). These pieces of middleware are packaged as compressed archives, ready for the deployment to the worker nodes.

- **Configuration files**: systems that are not integrated with a configuration framework often rely on files for their configuration. In the case of the SAM-Grid, the SAM system uses files to configure clients and servers. These files are created as part of the installation process of SAM. The SAM client is one of the VO-specific products that are installed at the gateway node.

These files are transferred to the worker nodes using the tools provided by the sandboxing framework. These tools provide the abstraction of a "sandbox" and methods to operate on it. Conceptually, the sandbox is an archive of files available at the gateway and that are needed at the worker node. The tools make available python and command line interfaces to operate on the sandbox. A sandbox can be created, local files can be added to it, it can be packaged, and it can be cleaned up. The relevant methods of the sandbox framework and how they are used to solve the four problems introduced above are discussed in the sections below.

#### 4.3.2.1 Information Transport

When running at the worker node of a cluster, a job relies on the availability of data transport clients to retrieve job environment information and data.

The usability of data transfer clients cannot be taken for granted because of the heterogeneity of native data transport mechanisms at the cluster and the potential unavailability of batch system stage-in capabilities and of a shared file system (sec. 4.3.1.1).

To overcome these problems, the SAM-Grid uses a technique called "sandbox packaging". The result of packaging a sandbox is creating a bootstrapping executable. The executable consists of a self-extracting archive, containing, among other things, the gridftp client as the data transfer protocol of choice. When the bootstrapping executable runs, it first extracts the data transfer archive, then it passes control to a "driver" script, which contains enough information about the sandbox to transfer all of its content to the worker node. The bootstrapping executable is submitted to the batch system as the executable program and transported to the worker node using native cluster mechanisms.

In more detail, the sandbox is created at the cluster gateway by the grid-to-fabric interface. The interface uses directives specified by the user in the job description file (sec. 3.2) to fill the sandbox with environment files specific for the job (configuration files, wrapper scripts, etc.). A set of standard environment files, such as the sam client middleware, the user credentials, etc., is also always added to the sandbox. Once the sandbox is ready, the grid-to-fabric interface packages it. At this time, a "driver" script is dynamically generated from a standard template. The driver contains the information on the full path of all the files in the sandbox, the value of standard SAM-Grid environment variables, such as JIM_GLOBAL_JID (the global job identification handler), and the command to execute after the sandbox has been transfered to the worker node. This command is generally a wrapper whose behavior depends on what type of job the user is executing. This information is also specified as part of the user's directives to the grid-to-fabric interface.

The driver is archived in a tar file, together with the gridftp client, the standard SAM-Grid GSI configuration for data transfers, the user credentials, and the fcp client. This archive is then transformed into a self-extracting executable by dynamically generating and compiling a C pro-

gram. This program contains the octal dump of the archive, represented as a large array. When executing at the worker node, the program writes the content of the array to the standard input of the tar command, then switches context by passing control to the driver. At this point, the driver can use the data transfer middleware (fcp and gridftp) together with the user credentials to transfer the content of the sandbox to the worker node, before passing control to the application. The driver relies on the presence of a gridftp server at the gateway node. Generally the server is run as part of the required services at the gateway node; alternatively, the sandbox framework can start up dynamically a gridftp server per job. While the latter solution makes the deployment of the gridftp service easier, since server instantiation is automatic, the former solution is more scalable and it is the one used in practice. In any case, the configuration of the server (host, port, and GSI identity) is saved in the driver script when the grid-to-fabric interface generates it.

Other grids solved the problem of environment preparation by relaxing some of the constraints that drove our design for the "information transport" problem (sec. 4.3.1.1). For example, LCG requires that clusters participating to the grid pre-install the gridftp client at every worker node (requirement 1). Open Science Grid, instead, assumes the presence of a shared file system between gateway and worker nodes (requirement 3). Despite the fact that these solutions violate the principle of site independence, both grids could negotiate these choices of resource configuration with the sites because of their political weight. The SAM-Grid does not have such weight, thus we had to find a solution that would adapt to pre-existing site configurations.

In summary, the sandbox framework solves the "information transport" problem using the following elements: a sandbox created by the grid-to-fabric interface; a self-extracting executable containing data transfer middleware, user credential, and a "driver" script; a data transfer server at the gateway node. The data transfer clients are made available at the worker node by the batch system, which transports them as a single self-extracting executable.

#### 4.3.2.2 Load Control

Transporting data transfer clients to the worker nodes using the sandbox framework has been used in production by the SAM-Grid for a year and a half on a dozen clusters. During this time, a drawback that was observed was the increase in the load of the gateway machine during sandbox packaging. This has been introduced in section 4.3.1.2 as the "load control" problem. The major contributors to the load are the tar and gcc commands, which are both CPU intensive. The increased load caused problems in the stability of the gateway machine when multiple grid jobs entered the cluster. This problem was addressed by programming the resource broker to throttle the number of jobs submitted to a gateway.

We start by describing a simplistic solution first, in order to illustrate the issues involved. In Condor, the resource classad (sec. 3.4) signals the availability of the resource to the broker using the *Requirements* attribute. This attribute is a boolean expression built using the values of other attributes: only when *Requirements* evaluates to *TRUE* will the resource accepts jobs. Job throttling was achieved by including in the resource classad three extra attributes:

1. *JobsEnteringTheCluster*: the number of grid jobs entering the cluster when the classad is advertised. The value of this attribute does not change at the information collector of the broker until a new classad is advertised. This number can be obtained by looking at the list of processes running at the gateway, where the advertisement framework runs.

2. *MaxEnteringJobsAllowed*: the maximum allowed number of jobs concurrently entering the cluster. This value is provided as part of the site configuration and depends on the computing power of the gateway machine.

3. *CurMatches*: this attribute is incremented by the broker every time a new job is matched to the cluster. With this criterion, the match maker considers a job as entering the cluster as soon as it is matched to

the cluster. Because of the delays between resource selection and job scheduling, this leads to a potential underutilization of the gateway. In practice, this was never a limiting factor. The value of *CurMatches* is initialized to *JobsEnteringTheCluster* when the classad is sent. *CurMatches* is a reserved name in the Condor framework.

The *Requirements* attribute used to throttle jobs is

$$Requirements = CurMatches < MaxEnteringJobsAllowed$$

This way, when the number of jobs assigned by the broker to the cluster exceeds the maximum number of jobs allowed by the site, the *Requirements* attribute evaluates to *FALSE* and the broker stops matching jobs to the cluster.

While in practice this technique solves the problem of the load to the gateway machine, it should be noted that it does not guarantee 100% accuracy. As mentioned, there is a time lag between matching the job to the cluster and scheduling it. If a new classad is sent to the broker during this time lag, the classad attributes *JobsEnteringTheCluster* and *CurMatches* will not have a record of the jobs scheduled to the cluster that have not yet entered. More jobs than the maximum allowed number of jobs may end up entering the cluster in this case, because nothing will prevent the jobs already scheduled from entering the cluster.

On the other hand, in some cases the total number of jobs entering the cluster may never be reached during an advertisement cycle, even if there are eligible jobs idle at the submission site. In fact, after the grid-to-fabric interface has finished submitting batch jobs to the batch system, the status of the grid jobs change from "entering" to "entered" the cluster. Because advertisements are sent periodically and not continuously, the broker may still consider jobs as "entering" the cluster, while in reality the jobs are "entered". This error is ultimately due to the time lag between two advertisement cycles. In this case, less jobs than the maximum number allowed will be matched to the cluster during the advertisement cycle. These inaccuracies, both stemming from time lags typical of the grid, tend to cancel each other and do not affect in practice the usability of the system.

In reality, the job throttling expression used in the SAM-Grid is more complicated than the one described above. The load to the head node, in fact, increases with the number of jobs running at the cluster, even if they are not in the status of entering the cluster. A control on the maximum number of grid jobs allowed at the cluster is also needed.

We list here the factors that contribute to the load of the gateway and that are due to the grid jobs that are running at (and not entering) the cluster. The GRAM protocol instantiates a process (job-manager) per grid job at the gateway. Because the SAM-Grid splits grid jobs into multiple instances of batch processes at the cluster, for a given number of batch processes it will have less job-managers running at the gateway node than other grids, where a grid job always corresponds to a batch job. Nevertheless, even in the case of the SAM-Grid, these processes running at the gateway contribute to the load on the gateway. In fact, as few as several dozen grid jobs, corresponding to several hundred batch processes, cause a load problem on the typical commodity machine,

Another contributing factor to the load is the periodic polling of the grid job status by the submission service. The grid job status is determined by aggregating the status of every batch job running in the batch system. The grid job is considered *done* when all the batch jobs are *done*. Information on the status of the batch jobs is gathered through the batch adapter layer (sec. 4.2) by querying the batch system. This process is in general also CPU intensive. Caching the statuses of the jobs is one way we control this load. When the batch system is queried, the returned statuses are recorded on a temporary file. Independently of the polling frequency of the submission site, we can control the load on the gateway node by returning the stale information contained in this file. The batch system is periodically queried again to refresh the statuses of the jobs. Caching techniques are used also by LCG and OSG. In particular, the latest version of the globus toolkit, GT4 [173], adopted by OSG in the deployment v0.4, implements caching using a relational database back-end.

Above a certain number of jobs, information caching is not a valuable solution anymore. Returning stale information slows the grid from knowing

the real status of the jobs. This increases the inefficiency of operations, especially if new jobs can be submitted only if other have already finished (sec. 3.2.4). In other words, the load on the gateway must be controlled not only by information caching, but also by limiting the total number of grid jobs running at the cluster. A way of achieving this is using job throttling techniques.

To implement job throttling to limit the number of jobs entering the cluster *and* the total number of jobs allowed, we advertise these attributes in the cluster classad:

1. *MaxJobsAllowed*: the maximum number of jobs that are allowed to be scheduled at a cluster. This value is configured as part of the site configuration.

2. *CurrentJobs*: the total number of jobs running on the system. This value is considered constant by the broker until a new classad is advertised. The advertisement framework obtains this number by looking at the list of processes running on the gateway machine.

3. *CurMatches*: this attribute is increased by the broker every time a new job is matched to the resource (see description above for *CurMatches*). To limit both the total number of jobs and the number of jobs entering the cluster, in this case *CurMatches* is initialized to *CurrentJobs*.

4. *JobsMatchedSinceLastAdvertisement*: this is a counter of the jobs that are matched to the resource since the last advertisement cycle. It is a positive number that is zero when the classad is first advertised and grows as the broker assigns jobs to the resource. The attribute is defined as $CurMatches - CurrentJobs$.

5. *JobsEnteringTheCluster*: the number of grid jobs entering the cluster when the classad is advertised. The value of this attribute does not change at the information collector of the broker until a new classad is advertised. This number can be obtained by looking at the list of processes running at the gateway, where the advertisement framework runs.

6. *MaxEnteringJobsAllowed*: the maximum allowed number of jobs concurrently entering the cluster. This value is provided as part of the site configuration and depends on the computing power of the gateway machine.

The *Requirements* attribute used to throttle jobs is

$$Requirements = (CurMatches < MaxJobsAllowed)\ \&\&\ (JobsMatchedSinceLastAdvertisement + JobsEnteringTheCluster) < MaxEnteringJobsAllowed$$

This expression prevents the broker from matching new jobs to the cluster in case the total number of jobs running on the cluster exceeds the maximum number of jobs allowed and in case the total number of jobs entering the cluster exceeds the maximum number allowed. The expression is the logical AND (&&) of two expressions. The first expression limits the number of total jobs running at the cluster. At every new advertisement cycle, *CurMatches* is initialized to the number of grid jobs currently running at the cluster (*CurrentJobs*) and increases every time the broker matches a job to the cluster. The expression becomes *FALSE* if *CurMatches* is equal to or greater than *MaxJobsAllowed*. The second expression limits the number of grid jobs entering the cluster at any time. The expression becomes *FALSE* when the number of grid jobs that were entering the cluster when the classad was sent (*JobsEnteringTheCluster*) plus the number of grid jobs matched by the broker to the cluster (*JobsMatchedSinceLastAdvertisement*) is equal to or greater than the total number of allowed jobs entering the cluster (*MaxEnteringJobsAllowed*).

A solution similar to our throttling solution has been adopted within the context of the Condor-G framework to do resource load balancing in a test bed grid for bio-informatics and it was demonstrated at the Super Computing 2003 Conference, Baltimore, by the Condor Team [174]. That grid consisted of a half dozen sites, each available to queue up to two dozens jobs. Even if no scalability limit due to high load was hit in that context, the demonstration was nevertheless a proof of principle of the functionality of the framework to deal with problems of high load.

We have also developed other scheduling rules to exercise a finer grain of control over job flow from the submission site. In particular, it turns out that packaging the output sandbox, at the end of the grid job, is also a CPU intensive operation. Algorithms have been created to stop the flow of jobs in case too many jobs are in the phase of entering the cluster and of packaging the output sandbox. These algorithms have not been used for production activities, since, in practice, we have never observed a machine crash due to these conditions.

In addition to invoking CPU intensive services, another contributor to the load of the gateway machine, as well as the data server machines, are the data transfers initiated by the batch jobs. This is another aspect of the "load control" problem (sec. 4.3.1.2). The SAM-Grid is particularly sensitive to this problem because of job aggregation management, that is the splitting of grid jobs at the gateway into multiple instances of batch processes (sec. 3.2). Because of aggregation management, typically dozens or hundreds of batch jobs tend to start approximately at the same time. The principal consequence of this for the SAM-Grid is that all the jobs try to concurrently transfer the sandbox to establish the job environment. Because the sandbox is available at the gateway node, the data transfer server at the gateway may be required to serve up to hundreds of data transfer requests at the same time. A typical dual 1 GHZ Pentium III CPU with 1 Gigabyte of RAM crashes at around 100 concurrent transfers. It should be noted that this problem not only affects the gateway when transferring the sandbox, but also storages when transferring data.

To overcome this problem, the SAM-Grid implements data access queues using a product called fcp [169]. The product consists of a client and a server programs. The fcp server is responsible for controlling access to the data transfer server. The server queues up client requests to transfer data and grants them $N$ at the time, where $N$ is a configurable parameter. In other words, the fcp client blocks until the server grants access, then executes the client data transfer command. In the case of the SAM-Grid, fcp servers are instantiated at the gateway node as well as at all the SAM data storage servers. In some cases, more than one data queue is also defined for the same

storage server (sec. 3.2). The administrators are responsible for configuring $N$ to protect their machines from crashing. Typically, for a modern dual CPU commodity machine, the administrators obtain a stable configuration by setting the server to allow half a dozen concurrent transfers.

#### 4.3.2.3 Service Configuration Accessibility

The SAM-Grid uses the local sandbox framework as the principal mechanism to establish the job environment at the worker node. No assumption is made regarding the pre-configuration of the worker node itself. The sandbox framework makes available at the worker node data transport clients using sandbox packaging ("information transport" problem, sec. 4.3.2.1) and provides load control of the data storage machines via data access queues ("load control" problem, sec. 4.3.2.2).

The SAM-Grid uses the sandbox framework to make available at the worker node the configuration files of those services that are not integrated with the SAM-Grid configuration infrastructure (sec. 2.1). The configuration of the SAM data handling system is an example of such service.

In practice, transporting the configuration files for these services to the worker nodes is not enough. The files must be installed, in order to enable fruition. In general, making available the configuration for these services is a two step process:

1. **Retrieval**: this step is done at the gateway node. It gathers the appropriate configuration files and make them ready for the transport to the worker node. When multiple configuration files for the same product are available, like in the case of SAM, this may not be a trivial operation.

2. **Installation**: this step is done at the worker node. It gets hold of the configuration file (already at the worker node) and prepares the job environment so that the interested services can make use of it. This typically implies installing the file at a specific location on the file system, or defining a service-specific environment variable pointing

to its path, or converting its content to a different format e.g. a set of environment variables.

Because these two steps are taken at different locations and, presumably, by different bodies of code, we could face the classical problem of keeping retrieval and installation instructions synchronized when code maintenance is needed. To minimize the risk of inconsistencies, we wanted to keep both instructions together in the same body of code.

The SAM-Grid provides a solution to this problem using a tool called product configuration manager (jim_manage_prod_config). The tool offers both a command line interface and a python API, and it is fully integrated with the sandbox framework, in order to provide information transport to the worker nodes. An initialization function allows the definition of both the retrieval and installation instructions. Invoking the *get_config* function, the retrieval instructions are executed and the resulting files added to the local sandbox. The installation instructions are appended to a control file, which is also added to the sandbox. The SAM-Grid grid-to-fabric interface uses this mechanism to retrieve the configuration of several products.

At the worker nodes, a job wrapper invokes the *set_config* function. The function automatically executes the commands in the control file, thus installing the configuration files for all the products. It should be noted that the code invoking the *set_config* function is always the same irrespectively of the products configuration retrieved at the gateway. This essentially eliminates the need for maintaining the piece of code executing at the worker node.

We use this mechanism not only to retrieve configuration files, but also to reduce the load on the configuration framework. By default, jobs access such configuration directly from the configuration framework database. When hundreds of jobs start concurrently at a cluster, we may incur in a database overload. This results in inefficient data access due to denial of service errors and relative access retrials by the jobs. As discussed in the solution to the "load control" problem (sec. 4.3.2.2), the sandbox framework provides mechanisms to ensure the scalability of the environment transport.

The product configuration manager and the sandbox framework can also be used to transport to the worker nodes the configuration of products integrated with the configuration framework. Once the information is at the worker node in the form of a file, it can be directly accessed from the file system, without accessing the site configuration database. This does not replace the use of the configuration framework, but gives us the option to preempt potential scalability limits using a solution designed to be scalable for hundreds of concurrently running jobs.

For the same reason of scalability, the sandbox framework is the technique of choice to distribute information common to a set of batch jobs, when this information is gathered at the gateway node. This is convenient, for example, when submitting montecarlo jobs to a site. The grid jobs are split typically into hundreds of local batch processes. Each process needs to access the SAM database to retrieve information about the montecarlo request at hand. The information is the same for all batch processes, since it is associated with the initial grid job. The grid-to-fabric interface can be instructed to pool this access and store the result in the sandbox in the form of a file. The sandbox framework will make the file available in the environment of the job, where the application can use it without contacting the SAM database directly. Ultimately, this mechanism reduces the load to the database machine (see example 1, sec. 3.2).

Figure 4.2 shows a real-life example of the use of the product configuration manager.

#### 4.3.2.4 Environment Uniformity

When running analysis jobs, physicists want to be able to use the same analysis framework over many months or years. This uniformity is necessary because physics results are often only comparable when obtained using the same version of the software. Rerunning older analyses with newer versions of the software is not done on a regular basis because of the computation time required.

When analysis jobs are run on dedicated resources, all necessary ver-

*Using the python API, from the code of the grid-to-fabric interface at the gateway node*

```
config = jim_manage_prod_config.JimManageConfigSettings(\
         ["--debug" ,\
          "--name=jim_job_managers",\
          "--get-cmd=. `ups setup jim_job_managers`;\
                     jim_config_manager_cmd.py gcs \
                       -n jim_job_managers > jim_job_managers_config",\
          "--set-cmd=export JIM_UPS_DB=`pwd`;\
                     mkdir -p $JIM_UPS_DB/jim_job_managers/etc;\
                     mv jim_job_managers_config
                        $JIM_UPS_DB/jim_job_managers/etc/config;\
                     export JIM_JOB_MANAGERS_DIR=\
                            $JIM_UPS_DB/jim_job_managers"])

jim_manage_prod_config.getConfig(config)
```

---

*Using the command line interface, from the job wrapper code at the worker node*

```
. `./jim_manage_prod_config.py set_all_config`
```

Figure 4.2: The code to retrieve the configuration of the jim_job_managers product at the gateway node (top) and to install the configuration in the job environment at the worker node (bottom). Both the retrieve and install commands are defined during the initialization of *JimManageConfigSettings* (*–get-cmd* and *–set-cmd* options respectively). Coded as a shell script, the retrieve command reads the configuration from the XML database, using its command line interface (jim_config_manager_cmd.py), and writes the configuration to the jim_job_managers_config file. The set command moves the same files to an appropriate location on the file system and defines an environment variable used by the product management system to locate the product. The set command is executed only at the worker node with the code shown at the bottom.

sions of the framework must therefore be pre-installed on the system. This is the responsibility of the system administrators and results in high maintenance load for the administrators themselves and for the users, who need to check if old versions are still available before submitting their jobs. Another problems is the inefficiency caused by several hundreds Gigabytes of disk space being permanently allocated to hold this software at all the dedicated clusters.

When analysis jobs are run on the grid, the above solution is not even feasible because software pre-installation is rarely allowed. Users can use RTE techniques to create a portable executable for their analysis. These techniques are implemented in a framework that recreates the Run Time Environment (RTE) of various DZero applications (sec. 4.3.1.4). The executable can then be included in the input sandbox when running on the grid. As with the software pre-installation solution, this has also drawbacks. The grid typically keeps the user input sandbox at several places throughout the lifetime of the job. For example, for the SAM-Grid it is kept at the Submission Service and at the Execution Site Gateway. Considering that lot of information is duplicated in different jobs, this leads to an inefficient use of the storage.

The solution adopted by the SAM-Grid consists in distributing the whole analysis framework via the SAM data handling system. The environment preparation framework is responsible for the dynamic installation of the software on the scratch area of the worker node before the job runs. For the DZero experiment, this translates in requiring 4 Gigabytes of local scratch disk space for every CPU at the worker nodes. This requirement is in practice always met.

A disadvantage of this approach, as compared to software pre-installation, is that jobs often waste CPU cycles waiting for the software to be transported to the worker node. In our experience, in fact, practically no cluster provides scalable local storage. In other words, when hundreds of jobs access data concurrently from the storage, the likelihood of having massive data transport failures and cluster downtimes is high, making the system not production-quality. In our solution, the software is retrieved via data access

queues that block the job until access to the data is granted. This minimizes problems of scalability of the storage, but it introduces inefficiencies in the usage of the computational resources. However, the same inefficiencies are also present when using "user provided executables". From this point of view, the SAM-Grid solution and the RTE techniques are essentially equivalent. In summary, because on the grid very few clusters allow software pre-installation and considering that RTE techniques are storage intensive, the SAM-Grid solution is the most favorable.

As compared to the other approaches, this solution has also the following advantages:

1. **Low administrative maintenance**: the software is delivered upon request to the site storage elements controlled by SAM. Because SAM is responsible for the disk space management, there is no longer a maintenance issue of the availability of older software versions at a site.

2. **Small size of the user input sandbox**: because the software is dynamically deployed at the worker node before control is passed to the application, the users effectively see the software as pre-installed. They can therefore limit their input sandbox to the file of macro directives to the framework (sec. 4.3). This reduces, by several orders of magnitude, the amount of disk space required, as compared to RTE techniques.

3. **Flexibility of the execution environment**: if necessary, our solution can be used in conjunction with user-provided RTE executables, providing a high degree of flexibility in configuring the job environment.

4. **Application cataloguing**: the software can be thoroughly catalogued using the metadata mechanisms of the SAM data handling system. This promotes the reproducibility of the physics results because not only the data, but also the binaries, can be tracked through the SAM system (sec. 1.3).

5. **Convenience of the distribution**: when a new release of the framework is available, storing it in the SAM system is the only action required for its publication. This makes the release immediately available, since no intervention is necessary from administrators to preinstall the software.

The user controls what software is deployed in the job environment by specifying in the job description file a SAM dataset definition name that corresponds to the list of binary files required by the job (JDL attribute *jobfiles_dataset*, sec. 3.2). Indicating a dataset name is a general mechanism to specify the products required by the job. Depending on the application type, the content of the dataset is in general different. For example, a binary dataset for the reprocessing application includes the DZero analysis framework and Runjob, the application management system (sec. 3.2.4). On the other hand, a dataset for the montecarlo production application contains, in addition to the analysis framework and Runjob, special framework configuration parameters that describe the physics of the events to simulate ("Card" files) and the characteristics of the accelerator magnetic field ("Magfield" files).

Despite its flexibility, though, this mechanism is arguably error prone. Because the files required are all specified in the job description as a single dataset name, important parameters, such as the version of the analysis framework, remain implicit. This may lead to user errors when trying to provide software uniformity. To overcome this problem, the user is required to specify the version of the framework software explicitly using the attribute *d0_release_version*. The client software, then, compares this information with the metadata of the framework file included in the binary dataset. The job is not submitted if the two information do not match.

The files identified by the binary dataset name are imported to the worker node by a job wrapper executing before the application. This wrapper uses SAM interfaces to trigger the delivery of the binary files to a SAM storage element close to the cluster. Once the files are there, the wrapper transports the files to the worker nodes using the data transport middle-

ware available in the job environment ("information transport" problem, sec. 4.3.1.1) and accessing the data queues of the SAM storage elements ("load control" problem, sec. 4.3.1.2). The information on these data queues is transported to the worker nodes as part of the local services configuration ("service configuration accessibility" problem, sec. 4.3.1.3). Once the binary files are at the worker nodes, the archives are deflated. This way, the software and its configuration has been dynamically deployed to the worker node. At this point, the wrapper passes control to the application, which can access all its necessary software products.

Other grids, such as LCG or OSG, do not provide support for dynamic installation of software products. LCG leaves this as a responsibility of the user, who needs to write wrappers around the applications in case software dependencies are needed. These dependencies can be uploaded to LCG storages before running the jobs. The jobs can then download them, using data transport middleware pre-installed at the worker nodes. OSG, instead, has negotiated with the resource providers the configuration of shared file system areas (called $DATA) where the user can deploy software before running the jobs. Once running at the worker nodes, the jobs can access this software via the shared file system. We do not like these solutions because they impose special requirements on the configuration of the clusters (middleware pre-installation and the presence of a shared file system, respectively), which violates the principle of distributed ownership of the resources.

It should be noted that the SAM-Grid solution to the "environment uniformity" problem (sec. 4.3.1.4) is transparent to the users and does not impose external requirements on the resource providers. Our solution relies on the solutions developed for the "information transport" problem (sec. 4.3.2.1), the "load control" problem (sec. 4.3.2.2), and the "service configuration accessibility" problem (sec. 4.3.2.3) and it is the last step required for the preparation of the job environment.

# Chapter 5

# SAM-Grid At Work and Conclusion

For the past two years, the SAM-Grid system has been used for data reconstruction and montecarlo production for the DZero experiment. In this chapter, we present the data reconstruction project and our conclusions.

## 5.1 Data Reconstruction for DZero

Throughout the lifetime of a high energy physics experiment, physicists continuously refine their understanding of the detector. Calibration constants are made more precise in order to measure with greater accuracy quantities such as position and energy of the particles. Processing algorithms are improved, for example, to define more precisely the trajectories of the particles or the spacial location of the event vertex. To take advantage of such improvements, experiments often reprocess the raw data a few times throughout the lifetime of the experiment. The reprocessed data are the basis for more precise data analyses and, eventually, scientific publications.

In 2004 the DZero experiment took the decision of reprocessing the entire data sample for the second time since the beginning of data taking in 1997. The data sample consisted of 250 Terabytes of raw data, comprising one billion physics events organized in 250,000 files. After studying the ap-

plication, it was estimated that the computing power necessary to reprocess the whole sample was 1,600 1GHz-CPU×years. The computing resources at Fermilab were already mostly allocated to process the raw data coming daily from the detector and to let physicists work on their analysis. The data reprocessing had to be done on remote resources. In October 2004, it was decided to use the SAM-Grid for the management of job, data, and information [175].

The data reprocessing project had two principal characteristics.

1. **Resource intensiveness**: the computational requirements of the project implied that the institutions participating in the project had to dedicate a substantial fraction of their resources continuously for a long period of time. Most institutions decided to appoint facility representatives to the project in order to ensure the efficient utilization and the fair usage accountability of their resources.

2. **Operational difficulty**: the project bookkeeping had to be conducted with an extreme level of accuracy. For completeness, all events in the sample had to be reprocessed, but no more than once, in order not to bias the physics results. To guarantee this level of accuracy, the load of the operations had to be spread among a group of people.

For these two reasons, the experiment assembled a team of half a dozen people, lead by two coordinators. This team was effectively the SAM-Grid customer throughout the duration of the data reprocessing project. The project is, to our knowledge, the largest data processing activity on the grid to date.

### 5.1.1 Challenges

The DZero reprocessing project presented several challenges for the team and the software infrastructure. These challenges are presented in this section. The solutions to the technical challenges are discussed in the appropriate sections, referenced below.

| | | |
|---|---|---|
| *DØFarm (Fermilab)*, | 1000CPUs | Also Traditional Computing |
| Lyon (France), | 400CPUs | Shared with LCG VO |
| Westgrid (Vancouver), | 600CPUs | Shared |
| U. Wisconsin (USA), | 30CPUs | Not a DØ institution |
| Prague (Czech Rep.), | 200CPUs | |
| DØSAR (UTA, Texas), | 200CPUs | |
| DØSAR (Oscer, Oklahoma), | 140CPUs | Shared with hurricane forecast |
| DØSAR (Sprace, Brazil), | 170CPUs | |
| CMS Farm (Fermilab), | 100CPUs | Same Gateway as OSG |
| London (UK), | 200CPUs | |
| GridKa (Germany), | 500CPUs | Shared by 8 HEP Experiments |
| Total (1GHz PIII CPU) | ∼3540CPUs | (2540 CPUs Remote) |

Table 5.1: The amount of computing resources available for the DZero reprocessing activity in 1GHz PentiumIII-equivalent processors.

**Coordination**: the operations of the team had to be coordinated in order to guarantee the reprocessing of the full data sample. One coordinator was responsible for giving periodically portions of the dataset to operators. New datasets were allocated to operators every couple of months, as already allocated datasets were processed. Most operators ran their datasets at their home institutions, so to guarantee full utilization of the resources and to have easy access to administrative control in case of failures.

**Short time**: the experiment desired to publish the data with the refined algorithms at the physics conferences held in 2006. Therefore, the deadline for completing the project was the end of 2005. Starting from October 2004, this gave the project about 6 months of development and deployment of the infrastructure and a subsequent 6 months of data production. The latter was achievable, considering that the total number of 1GHz-equivalent CPU available to the experiment throughout the collaboration was about 3,300 and that the estimated required CPU was 1,600 1GHz-CPU×years. Table 5.1 shows the details of the number of resources available for the DZero reprocessing project throughout the collaboration.

**Large deployment**: a big portion of the effort to prepare the infrastructure was the deployment of the SAM-Grid. This overlapped with the

beginning of the reprocessing, started in March 2005, because of constraints related to hardware procurement for some of the facilities. The 3,300 1GHz-equivalent CPU, in fact, were available as fractions of about a dozen computing centers throughout the US, Europe, and Latin America. The facilities that were not already part of the SAM-Grid have been brought online through May 2005.

As new sites were made available to the grid and as the group became familiar with the operations, the number of jobs run per day increased to a steady plateau of 4,000 jobs per day (fig. 5.1). This rate of job submission corresponded to the expected average production of six million events per day, to complete the reprocessing of the 1 billion events sample in 6 months. The plot of figure 5.1 shows also the slow down of operations toward the end of the project (September 2005), as operators finished their allocated datasets and stopped using their facility for data reprocessing.

Operators local to large facilities were made responsible for the deployment of the SAM-Grid infrastructure at their site. A central team of experts, located at Fermilab, was actively supporting the operators with the deployment and subsequent necessary optimizations. Even if arguably not most efficient, this model contributed to the familiarization of the operators with the SAM-Grid and soon built up an active geographically spread support community. Following this model, the infrastructure could be typically deployed in a few weeks. This included the installation and configuration of the software, the testing, and the cluster "certification" for production.

Cluster "certification" was particularly important to give the team confidence that the output produced at every cluster was physically sound. To this end, plots of physical quantities produced by the newly deployed sites were compared with a standard reference, produced with traditional (non-grid) computational mechanisms. To reduce the time required to generate the certification plots, only 10% of the events in the reference sample was used. The plots were inspected by a physicist who certified the consistency of the results within the expected statistical variance (fig. 5.2).

**Full resource utilization**: in order to complete the project in the

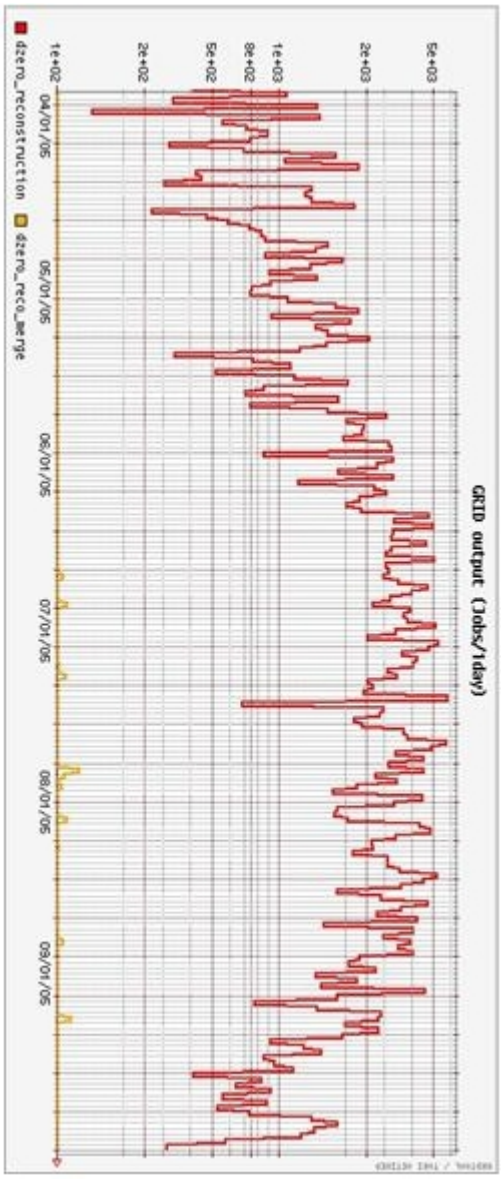


Figure 5.1: The number of DZero reprocessing jobs running on the SAM-Grid every day.

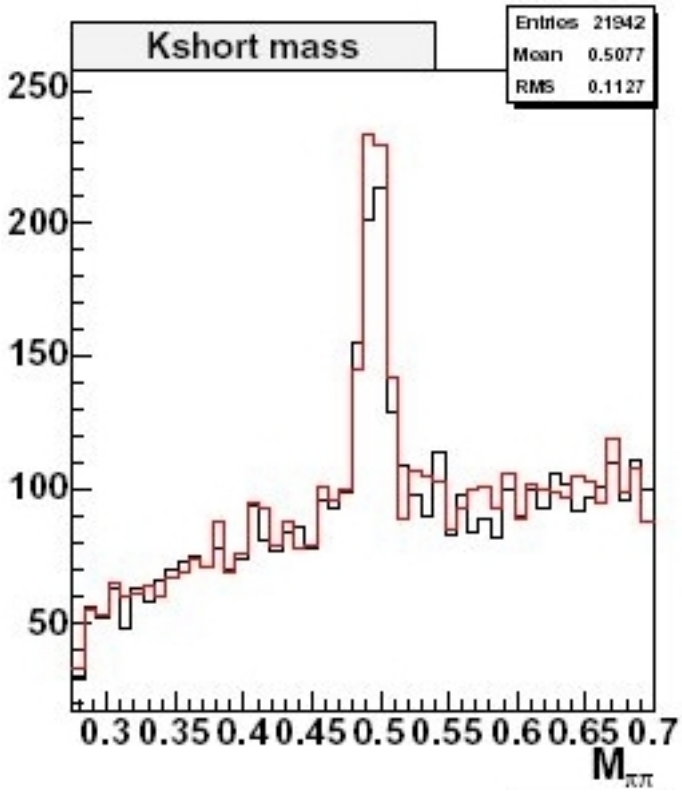


Figure 5.2: The mass plots of Kshort particles from the DZero detector. The red and black traces are generated from standard farm and SAM-Grid processing mechanisms respectively. The traces are statistically equivalent.

estimated 6 months, the infrastructure had to continuously fully utilize the DZero share of resources available at collaborating institutions. In addition, the system had to be able to use opportunistically all available resources at a cluster, in case the activity of other groups ceased temporarily. Considering the size of available clusters, this meant supporting up to 1,000 concurrently running jobs at the same site. This was particularly challenging since the standard middleware supported a maximum of 200 concurrent jobs on a modern commodity CPU (sec. 4.1). The solution to this scalability problem is discussed in chapter 4.

**Large data volumes**: Raw and derived data from the DZero experiment is stored at Fermilab. During the reprocessing activity, 75% of the physics events were processed at institutions outside Fermilab. This meant moving about 190 TBytes of data to "remote" facilities, in order to take advantage of the computing power available at the collaborating sites. Data movement was managed by the SAM data handling system (sec. 1.3).

Figure 5.3 shows a bar diagram of the number of events produced at each site.

**Job recoverability**: In a system as complex as the SAM-Grid, job failure is a daily occurrence. Failures are due to problems in grid services (SAM, job management services, etc.), local services (batch system, local storage systems, etc.), and hardware (worker and key service node crashes, file system failures, etc.). Rather than concentrating our efforts on the tuning of the infrastructure at each site to minimize the rate of failure, we decided to accept a rate of up to 15% and focus our effort on developing a reliable recovery mechanism for our data processing model. We now describe this model.

The raw detector data, input to the reprocessing software, is naturally organized into *daysets*, where each dayset is the set of files acquired from the detector in a day. The number of files in the dayset depends on the average daily conditions of the accelerator (beam luminosity) and on the detector trigger configuration (online event selection criteria). A typical DZero dayset consists of approximately 100 files. A dayset can be trivially

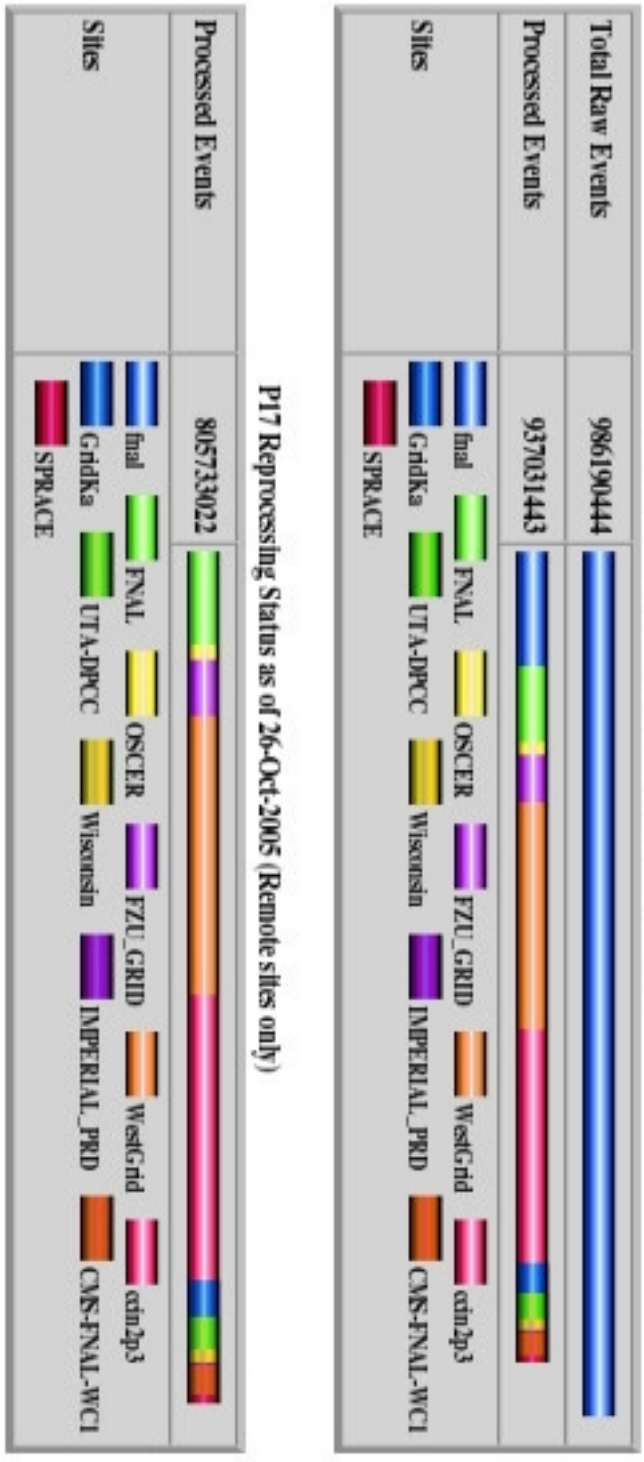


Figure 5.3: The number of data reprocessing events produced on the SAM-Grid per site. 75% of the events were processed outside Fermilab and less than 20% using traditional computation mechanisms (blue bar tagged "fnal").

represented in terms of a SAM *dataset definition* i.e. a named query in the SAM file metadata catalogue.

Every couple of months, each institution was assigned new daysets to reprocess. Operators treated the dayset as a unit of processing and expected the SAM-Grid to allow for such model. To comply with such expectations, SAM-Grid grid jobs for the data reprocessing application were designed to accept a SAM dataset definition as input. In this way, operators could assign daysets to grid jobs, making a grid job the unit of data processing.

On a modern 3 GHz commodity CPU, the DZero reprocessing application processes a typical raw data file, 1 Gigabyte in size, in about 6 hours. If a grid job could be split at a cluster into as many batch processes as files in its dayset, the grid job would typically take less than a day to complete, even on the smallest of our clusters. Depending on the size of the cluster, an operator could then submit a number of grid jobs to the cluster and check what jobs succeeded or failed the next day. The timing of this cycle would match very well with the daily routine of the operators.

In addition to the operational convenience, processing a single file per batch job also facilitates the recovery procedures. In fact, if a batch job fails, it is trivial (with appropriate bookkeeping) to identify what file must be reprocessed.

For these reasons, we decided to split grid jobs into a number of batch processes that correspond to the number of files in the input dataset. The multiplicity of the batch processes is managed by the grid-to-fabric interface when the grid job enters the site (sec. 4.1).

It turns out that the output of each batch job cannot be immediately used for analysis consumption. The size of a typical output file, in fact, is 200 Megabytes, too small for efficient storage in the mass storage system of the experiment (sec. 3.2.4). The processed files, therefore, must be merged into larger files, typically 1 Gigabyte in size. This is accomplished by submitting a grid job running a "merge" application. The output of the reprocessing jobs i.e. the input to the merging application, is stored in SAM on volatile storage, in what we call *durable locations*. The merging application retrieves these files via SAM and, after merging them, stores the "large" output to

the mass storage system at Fermilab, again via SAM.

This processing model, consisting of a production and a merging phases, completely relies on SAM for the bookkeeping of the data. In fact, input and output from each phase is handled via SAM. The SAM system, therefore, has a record of the whole processing history and parentage of the files. On this fact, actually, we have based our job recovery mechanism.

In this processing model, a job is considered failed if SAM does not have a record of its output. Therefore, to recover a processing job, we follow the following steps. After a grid job finished [1], we iterate through each raw file in its input dayset. For each raw file, we lookup its reprocessed "child" file in the SAM file metadata catalogue. If such "child" file does not exist, the raw file must be reprocessed. We thus build the recovery dataset, using the list of non-processed raw file names to create a SAM dataset definition. This dataset can be submitted as the input to a recovery reprocessing grid job. The recovery procedure is then followed again at the end of each recovery job, until the dayset is completely processed.

A similar procedure is followed to recover failed grid merging jobs. In this case, the procedure acts on the input to the merging job i.e. the dataset of reprocessed files generated from a certain dayset. As for the grid processing jobs, we iterate through the list of files in the job input dataset. For each reprocessed file, we lookup its merged "child" file in the SAM file metadata catalogue. If this "child" file does not exist, the reprocessed file must be re-merged.

The experiment has developed a series of tools to help with the operations. Such tools include the routines to check the completeness of jobs and to create recovery jobs.

### 5.1.2 Failure Analysis

The bulk of the reprocessing activity finished in seven months, at the end of October 2005. Figure 5.4 shows the plot of the integrated number of

[1] This recovery procedure should be followed 12 hours after the job is finished, as, in case of failure, the SAM file storage server retries to store output files for 12 hours, before timing out.

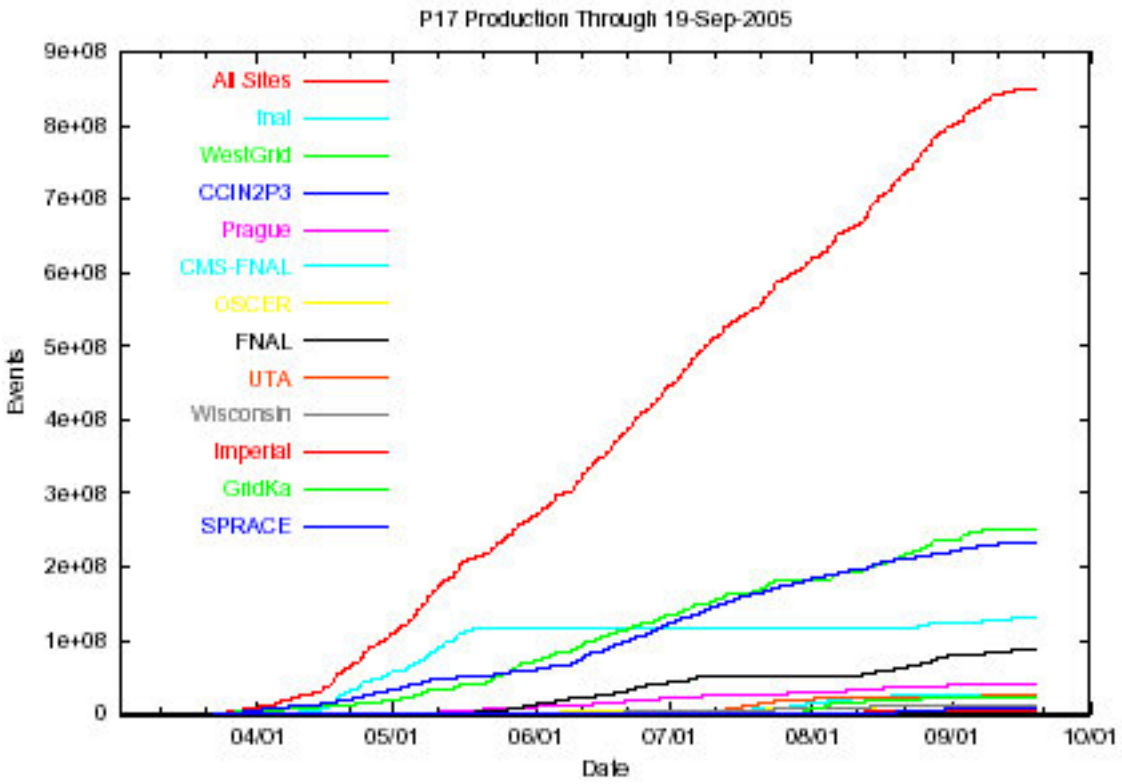


Figure 5.4: The integrated number of DZero reprocessing events produced as a function of time.

events processed as a function of time. The job success rate averaged 85% throughout the grid during stable operations (after middle of May), as shown in figure 5.5, right plot. Failed jobs are defined as those that did not hand over their output to the SAM system. It should be noted that this definition of job "efficiency" is not directly related to the resource usage efficiency. Many job failures, in fact, are immediate and constitute a minimal waste of computing resources. Examples of this type of failures include problems with the job environment preparation framework (sec. 4.3).

The left plot of figure 5.5 shows a possible categorization of the types of job failures. The information in the plot is derived from the analysis of the data in the job monitoring infrastructure (sec. 2.2.3). This analysis is based on the exit code of the grid wrapper around the application (sec. 4.3), as reported to the monitoring. Three categories of failures can be studied with this mechanism.

1. **No Exit Code**: these are the jobs that did not report any exit status to the monitoring system. In the plot, these are represented by the black trace (tagged *failed_no_code*). It is not immediate to distinguish among the reasons why this happens. These reasons are describe hereby.

   (a) *Failures of the monitoring system*: the XML database that was used to store monitoring information (sec. 2.2.3) was not available or the connection from the client failed. In principle, these jobs can be successful even if the monitoring does not report it. In this case, one could in principle identify this as a monitoring system failure, because a successful job hands over its output to the SAM system and, therefore, the bookkeeping service has a record about it.

   (b) *Failures of the job environment preparation*: these are fast-fail conditions, which waste a minimal amount of resources. If the job environment cannot be established properly, the monitoring client software may not be available to the job.

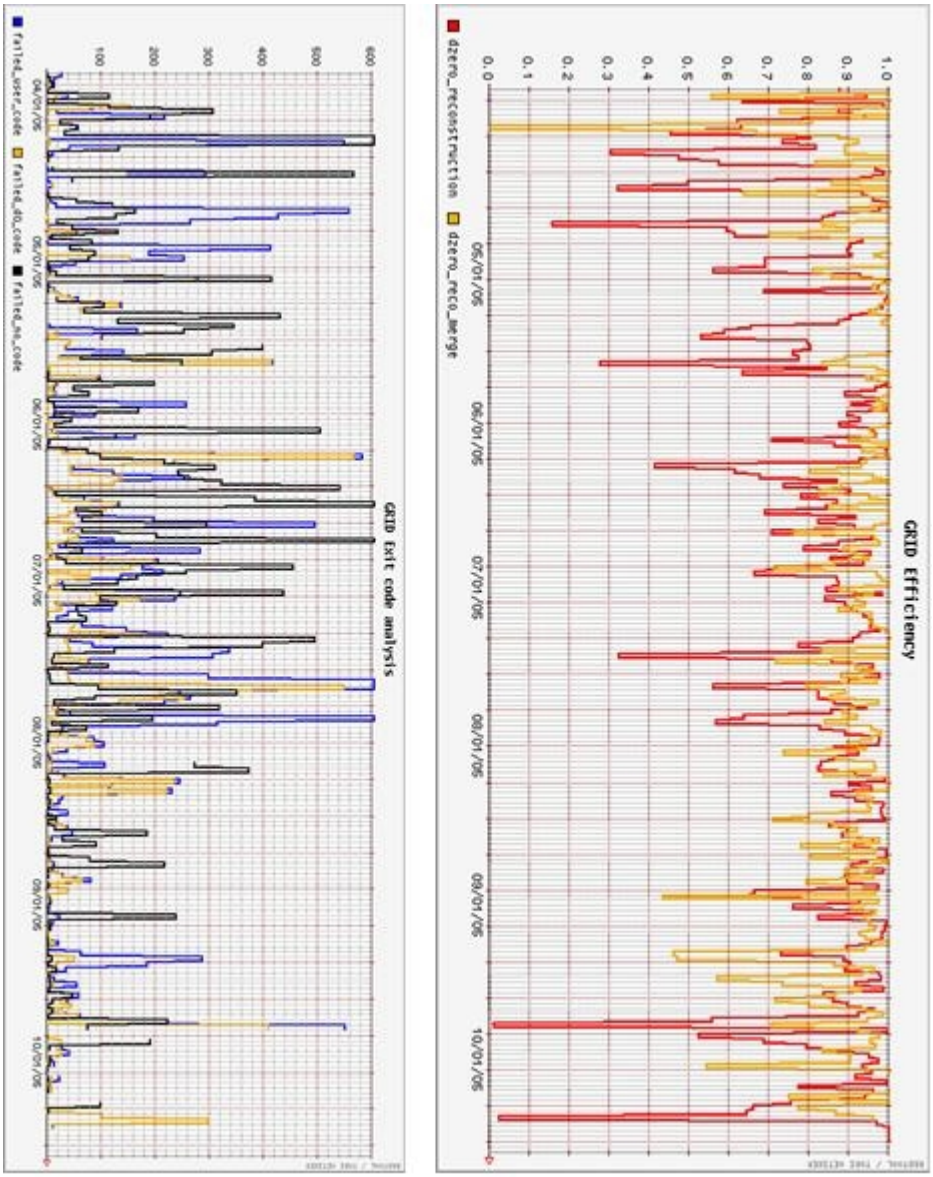


Figure 5.5: **Right Plot**: The success rate of local reprocessing jobs running on the SAM-Grid (daily binning). **Left Plot**: Analysis of the number of jobs failed per day (top $Y$ axis set to 600 failed jobs, to show structure).

(c) *Killed batch process*: the batch system kills the jobs, thus the monitoring notification routines are not executed. This can happen, for example, because of administrative actions e.g. cluster maintenance, or because the job has violated some of the system execution policy e.g. surpassed maximum allowed execution time, disk quota, etc.

(d) *Hardware failure*: worker nodes crashes e.g. components faults, power outages, etc.

2. **Grid Wrapper Failure**: the grid wrapper interacts with services of the grid on behalf of the application (sec. 4.3). This is needed, since practically all DZero applications are not grid-aware. The wrapper passes control to the application management system (sec. 4.1), which controls the execution of the application. If either the wrapper or the application management system fail, the job sends a notification message to the monitoring system. In this case, the application has certainly failed because of problems with the infrastructure, rather than because of application bugs. These types of failures are represented in the plot by a blue trace (tagged *failed_user_code*). A typical reason for these failures are problems with the data handling system, either at the level of global or local services. An example of a problem with a global service is the inability to contact the central SAM database. In the plot, this is sometime due to jobs submitted during the scheduled monthly downtime of the SAM database (first Tuesday of every month). These failures tend to be catastrophic. Since DZero reprocessing is a data intensive application, the interaction with the data handling system is indispensable to the execution of the job. If SAM has problems, all batch jobs submitted to the grid will probably sooner or later fail. An example of a data handling problem with local services are failures in the local data storages. The file may be available in the storage, but the data transport service (gridftp, rfio, etc.) may be unavailable.

3. **Application Failure**: These failures are due to bugs in the applica-

| Westgrid Production Summary | |
|---|---|
| RawFiles | 106178 |
| Daysets | 1636 |
| GridJobs | 3331 |
| BatchJobs | 134745 |
| FailedBatchJobs | 28567 |
| CorruptedFiles | 680 |
| BatchJobs/RawFiles | 1.269 |
| RawFiles/GridJobs | 31.876 |
| FailedBatch/BatchJobs | 0.2120 |
| CorruptedFiles/RawFiles | 0.0064 |
| GridJobs/Daysets | 2.0361 |

Table 5.2: DZero production statistics at the westgrid site at the end of the activity.

tion code, to corruptions of the input file, or to the unavailability of the local calibration databases. The failures are represented by the yellow trace (tagged *failed_d0_code*) and, because of the way the plot is generated, also contribute to the number of infrastructural failures (blue trace). By the end of the reprocessing activity, it turned out that 0.2% of the files in the full dataset generate unrecoverable failures, due to data corruption. Recovery of these files is sometimes possible with special care.

Table 5.2 shows statistics of the data reprocessing activity at Westgrid, one of the largest clusters in the SAM-Grid.

## 5.2 Conclusion

The SAM-Grid is a large distributed meta-computing infrastructure for DZero and CDF, two high energy physics experiments taking data at the Fermi National Accelerator Laboratory, Batavia, Illinois. The system is architecturally divided into three functional components: job and information management components, based on the standard grid middleware, and

the data handling component, implemented by the Sequential Access via Metadata (SAM) system. The design and implementation of the SAM-Grid job and information management components are the main subject of this research.

All components of the SAM-Grid offer a set of global and local services. These interact with the computational, storage, and network resources (the *fabric*) of the collaborating institutions, via a grid-to-fabric interface. The SAM-Grid implementation of this interface enables a robust interaction of the global services with a wide range of local services and resources (data handling, monitoring, job environment preparation, local job scheduler, etc.), using extensible and flexible adaptation layers.

The SAM-Grid mechanism of interfacing the grid to the fabric allows local system administrators to choose the resource configuration and local management software that best suits the needs of each site. Thus, the SAM-Grid promotes site autonomy and fosters the distributed ownership of the resources, the basic paradigms of grid computing. This is one major contribution of this work.

In order to maximize data throughput, the SAM-Grid optimizes the usage of local resources by coordinating the resource interaction with local and grid services. Resource and service coordination is achieved through a set of services that are aware of requirements and resource usage patterns of principal high energy physics applications. This is another major contribution of this research.

In the past year alone, the SAM-Grid has processed hundreds of Terabytes of data, running thousands of jobs every day on dozens of participating institutions throughout the world. We believe that this is the largest data processing activity ever attempted on a data grid system to date. This, more than anything, proves the scalability, flexibility, and high throughput of the SAM-Grid system.

# Bibliography

[1] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International J. Supercomputer Applications, 15(3), 2001.

[2] The BaBar Collab., D. Boutigny et al., "Technical Design Report", SLAC-R-95-457.

[3] The BELLE Collab., M. T. Cheng et al., "Technical Design Report", KEK-Report 95-1.

[4] The D0 Collab., "The D0 Upgrade: The Detector and its Physics", Fermilab Pub-96/357-E.

[5] CDF Collab., R. Blair et al., "The CDF II Detector Technical Design Report", FERMILAB-Pub-96/390-E.

[6] D. G. York, et al., "The Sloan Digital Sky Survey: Technical Summary", The Astronomical Journal 120 (2000) 1579-1587

[7] Bruce Allen, et. al., "Determining Upper Limits on Event Rates for Inspiralling Compact Binaries with LIGO Engineering Data", LIGO technical report T010025-00-Z (2001).

[8] CERN: http://www.cern.ch/

[9] The CMS Collaboration, "The Compact Muon Solenoid Technical Proposal", CERN/LHCC 9438, LHCC/P1 1994

[10] The CMS Collaboration, "Computing Technical Proposal", CERN/LHCC 96-45, (Geneva 1996)

[11] P. Saiz, L. Aphecetche, P. Buncic, R. Piskac, J. E. Revsbech and V. Sego, "AliEn - ALICE environment on the GRID", Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, Volume 502, Issues 2-3 , 21 April 2003, Pages 437-440

[12] The ALICE Collaboration, "ALICE Technical Proposal for A Large Ion Collider Experiment at the CERN LHC", CERN/LHCC/95-71, 15 December 1995

[13] The LHCb Collaboration, "LHCb: Technical Proposal", CERN-LHCC-98-004;

[14] The Atlas Collaboration, "Atlas - Technical Proposal", CERN/LHCC94-43, CERN, December 1994.

[15] Global Grid Forum: http://www.ggf. org/

[16] Globus Alliance: http://www.globus.org/

[17] I. Foster and C. Kasselman, "Globus: A Metacomputing Infrastructure Toolkit", International Journal of Supercomputer Applications, 11(2): 115-128, 1997

[18] A.S. Grimshaw, A. Natrajan, M.A. Humphrey and M.J. Lewis, A. Nguyen-Tuong, J.F. Karpovich, M.M. Morgan, A.J. Ferrari, "From Legion to Avaki: The Persistence of Vision", Grid Computing: Making the Global Infrastructure a Reality, eds. Fran Berman, Geoffrey Fox and Tony Hey, 2003.

[19] Platform Computing, "PLATFORM GLOBUS TOOLKIT: Open-source, commercially supported toolkit for building grids", On line http://www.platform.com/products/Globus/

[20] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: Architecture and Performance of an Enterprise Desktop Grid System", Journal of Parallel Distributed Computing, Vol 63, Issue 5, May 2003, pages 597-610.

[21] Sun Grid Engine: http://wwws.sun.com/software/gridware

[22] United Devices: http://www.uniteddevices.com/

[23] Parabon: http://www.parabon.com

[24] ProcessTree: http://www.processtree.com/ , Distributed Science Inc, Nov. 2000.

[25] Popular Power: http://www.PopularPower.com/

[26] Mojo Nation: http://www.mojonation.net/

[27] DataSynapse: http://www.datasynapse.com/

[28] R. Buyya, D. Abramson, and J. Giddy, "Nimrod-G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid", The 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), May 2000, Beijing, China, IEEE Computer Society Press, USA.

[29] F. Berman and R. Wolski, "The AppLeS Project: A Status Report, Proceedings of the 8th NEC Research Symposium", Berlin, Germany, May 1997.

[30] J. Novotny, "The Grid Portal Development Kit", Concurrency: Practice and Experience 2000; 00:1-7

[31] G. Allen, T. Dramlitsch, I. Foster, T. Goodale, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen, "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus",in Proceedings of Super Computing 2001, Nov. 2001, Denver, Colorado.

[32] J. Basney and M. Livny, "Deploying a High Throughput Computing Cluster", High Performance Cluster Computing, R. Buyya (editor). Vol. 1, Chapter 5, Prentice Hall PTR, May 1999.

[33] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids", in Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10), IEEE CS Press, Aug. 2001.

[34] I. Foster, "Grid Technologies & Applications: Architecture & Achievements", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP01), Beijing, China, Sep. 2001

[35] J. Moore, "Portals could lower grid barriers", Federal Computer Week, Oct 2003

[36] W. Hoschek, J. Jean-Martinez, A. Samar, H. Stockinger, K. Stockinger, "Data Management in an International Data Grid Project", 1st IEEE / ACM International Workshop on Grid Computing (Grid 2000), Bangalore, India, Dec. 2000.

[37] J. Apostolakis, G. Barrand, R. Brun, P. Buncic, V. Innocente, P. Mato, A. Pfeiffer, D. Quarrie, F. Rademakers, L. Taylor, C. Tull, T. Wenaus, "Architecture Blueprint Requirements Technical Assessment Group (RTAG)", Report of the LHC Computing Grid Project, CERN, Oct. 2002

[38] P. Buncic, F. Rademakers, R. Jones, R. Gardner, L.A.T. Bauerdick, L. Silvestris, P. Charpentier, A. Tsaregorodtsev, D. Foster, T. Wenaus, F. Carminati, "LHC Computing Grid Project: Architectural Roadmap Towards Distributed Analysis", CERN-LCG-2003-033, Oct-2003

[39] P. Eerola, B. Konya, O. Smirnova, T. Ekelof, M. Ellert, J.R. Hansen, J.L. Nielsen, A. Waananen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter, "The nordugrid production grid infrastructure, status and plans", in Proceedings of the 4th

International Workshop on Grid Computing, pages 158-165. IEEE CS Press, 2003.

[40] EGEE - Enabling Grids for E-science in Europe: http://egee-intranet.web.cern.ch/egee-intranet/gateway.html

[41] E. Hjort, J. Lauret, D. Olson, A. Sim, A. Shoshani, "Production mode Data-Replication framework in STAR using the HRM Grid", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP04), Interlaken, Switzerland, Oct. 2004

[42] I. Foster, J. Vckler, M. Wilde, Y. Zhao, "Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation", in Proceedings of 14th International Conference on Scientific and Statistical Database Management (SSDB '02), Edinburgh, July 2002.

[43] The Grid2003 Project, "The Grid2003 Production Grid: Principles and Practice", iVDGL, Technical Report, 2004: On line www.ivdgl.org/grid2003.

[44] The Particle Physics Data Grid, "GRID2003 Lessons Learned", PPDG Document 37, http://www.ppdg.net

[45] L. Pearlman, C. Kesselman, S. Gullapalli, B.F. Spencer, Jr., J. Futrelle, K. Ricker, I. Foster, P. Hubbard, C. Severance, "Distributed Hybrid Earthquake Engineering Experiments: Experiences with a Ground-Shaking Grid Application", in Proceedings of the 13th IEEE Symposium on High Performance Distributed Computing (HPDC-13), 2004.

[46] SAM-Grid project: http://www-d0.fnal.gov/computing/grid

[47] I. Terekhov, A. Baranovski, G. Garzoglio, A. Kreymer, L. Lueking, S. Stonjek, F. Wuerthwein, A. Roy, T. Tannenbaum, P. Mhashilkar, V. Murthi, R. Walker, F. Ratnikov, T. Rockwell, "Grid Job and Information Management for the FNAL Run II Experiments", in Proceedings

of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, Ca, USA, March 2003.

[48] R. Walker, A. Baranovski, G. Garzoglio, L. Lueking, D. Skow, I. Terekhov, "SAM-GRID: A System Utilizing Grid Middleware and SAM to Enable Full Function Grid Computing", in Proceedings of the 8th International Conference on B-Physics at Hadron Machines (Beauty 02), Santiago de Compostela, Spain, Jun. 2002

[49] G. Garzoglio, A.Baranovski, H. Koutaniemi, L. Lueking, S. Patil, R. Pordes, A. Rana, I. Terekhov , S. Veseli, J. Yu, R. Walker, V. White, "The SAM-GRID project: architecture and plan.", talk at the 8th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT-02), Moscow, Russia, Jun. 2002, Nuclear Instruments and Methods in Physics Research, Section A, NIMA14225, vol. 502/2-3 pp 423 - 425

[50] I. Terekhov et al., "Meta-Computing at D0"; talk at the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT-02), Jun. 2002, Nuclear Instruments and Methods in Physics Research, Section A, NIMA14225, vol. 502/2-3 pp 402 - 406

[51] M. Burgon-Lyon et al., "Experience using grid tools for CDF Physics"; talk at the IX International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT-03), Tsukuba, Japan, Dec 2003; to appear in Nuclear Instruments and Methods in Physics Research, Section A

[52] Fermi National Accelerator Laboratory: http://www.fnal.gov/

[53] PPDG: http://www.ppdg.net/

[54] GridPP: http://www.gridpp.ac.uk/

[55] SAM project: http://d0db.fnal.gov/sam

[56] L. Loebel-Carpenter, L. Lueking, C. Moore, R. Pordes, J. Trumbo, S. Veseli, I. Terekhov, M. Vranicar, S. White, V. White, "SAM and the particle physics data grid", in Proceedings of Computing in High-Energy and Nuclear Physics. Beijing, China, Sep 2001.

[57] I. Terekhov, R. Pordes, V. White, L. Lueking, L. Carpenter, J. Trumbo, S. Veseli, M. Vranicar, S. White, H. Schellman, "Distributed Data Access and Resource Management in the D0 SAM System", in Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10), San Francisco, California, Aug. 2001

[58] I. Terekhov, V. White, L. Lueking, L. Carpenter, H. Schellman, J. Trumbo, S. Veseli, and M. Vranicar, "SAM for D0-a fully distributed data access system", talk at Advanced Computing And Analysis Techniques In Physics Research (ACAT 2000) Batavia, Illinois, Oct 2000, in American Institute of Physics (AIP) Conference Proceedings Vol 583(1) pp. 247-249. August 20, 2001

[59] V. White et al., "D0 Data Handling", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP01), Beijing, China, Sep. 2001

[60] L. Carpenter et al., "SAM Overview and Operational Experience at the D0 experiment", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP01), Beijing, China, Sep. 2001

[61] L. Lueking et al., "Resource Management in SAM and the D0 Particle Physics Data Grid", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP01), Beijing, China, Sep. 2001

[62] L. Lueking et al., "The Data Access Layer for D0 Run II"; in Proceedings of Computing in High Energy and Nuclear Physics (CHEP 2000) Padova, Italy, Feb. 2000

[63] A. Rajasekar, M. Wan, R. Moore, W. Schroeder, G. Kremenek, A. Jagatheesan, C. Cowart, B. Zhu, S.Y. Chen, R. Olschanowsky, "Storage

Resource Broker - Managing Distributed Data in a Grid", Computer Society of India Journal, Special Issue on SAN, Vol. 33, No. 4, pp. 42-54 Oct 2003.

[64] H. Stockinger, A. Samar, S. Muzaffar, and F. Donno, "Grid Data Mirroring Package (GDMP)", Scientific Programming Journal - Special Issue: Grid Computing, 10(2):121-134, 2002.

[65] H. Stockinger, F. Donno, E. Laure, S. Muzaffar, P. Kunszt, G. Andronico, P. Millar, "Grid Data Management in Action: Experience in Running and Supporting Data Management Services in the EU DataGrid Project", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, Ca, USA, March 2003

[66] A. Chervenak, E. Deelman, I. Foster, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, H. Stockinger, K. Stockinger, and B. Tierney, "Giggle: A Framework for Constructing Scalable Replica Location Services", in Proceedings of the International IEEE Supercomputing Conference (SC 2002), Baltimore, USA, November 2002.

[67] J. Bent et al., "Flexibility, Manageability, and Performance in a Grid Storage Appliance", in Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), Edinburgh, Scotland, Jul. 2002

[68] W. Deng, T. Wenaus, "Magda - Manager for grid-based data", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, California, March 2003

[69] M. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, J. Saltz, "Distributed Processing of Very Large Datasets with DataCutter", Parallel Computing, 2001, pp. 1457-1478.

[70] V. Raman, I. Narang, C. Crone, L. Haas, S. Malaika, T. Mukai, D. Wolfson, C. Baru, "Services for Data Access and Data Processing on Grids", Global Grid Formum Document GFD-I.14 Feb. 2003.

[71] I. Terekhov, "SAMGrid Experiences with the Condor Technology in Run II Computing", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP04), Interlaken, Switzerland, Sep 2004.

[72] A.Tsaregorodtsev, V.Garonne, J. Closier, M. Frank, C. Gaspar, E. van Herwijnen, F. Loverre, S. Ponce, R.Graciani Diaz, D. Galli, U. Marconi, V. Vagnoni, N. Brook, A. Buckley, K. Harrison, M.Schmelling, U.Egede, A. Bogdanchikov, I. Korolko, A. Washbrook, J.P.Palacios, S. Klous, J.J.Saborido, A. Khan, A.Pickford, A. Soroko, V. Romanovski, G.N. Patrick, G.Kuznetsov, M. Gandelman, "DIRAC - Distributed Infrastructure with Remote Agent Control", in Proceedings of Computing in High Energy and Nuclear Physics, La Jolla, California, March 2003

[73] K. Harrison, W. T. L. P. Lavrijsen, C. E. Tull, P. Mato, A. Soroko, C. L. Tan, N. Brook, R. W. L. Jones, "GANGA: a user-Grid interface for Atlas and LHCb", in Proceedings of Computing in High Energy and Nuclear Physics, La Jolla, California, March 2003

[74] G. Avellino et al., "The EU DataGrid Workload Management System: towards the second major release", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, California, March 2003

[75] S. Fisher, "Relational model for information and monitoring", technical report GWD-Perf-7-1, GGF, 2001

[76] A. Iamnitchi and I. Foster, "A Peer-to-Peer Approach to Resource Location in Grid Environments", In J. Weglarz, J. Nabrzyski, J. Schopf, and M. Stroinski eds. Grid Resource Management, Kluwer Publishing, 2003.

[77] GridWeaver Technical Reports: http://www.epcc.ed.ac.uk/gridweaver/docs/

[78] P. Goldsack, "SmartFrog - a framework for configuration", Talk at the Large Scale System Configuration Workshop, Nov. 2001

[79] P. Anderson et al, "Towards automation of computing fabrics using tools from the fabric management workpackage of the EU DataGrid project", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, Ca, USA, March 2003.

[80] J.P.Wellisch, C. Williams, and S. Ashby, "SCRAM: Software configuration and management for the LHC Computing Grid project.", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, Ca, USA, March 2003.

[81] The Apache Software Foundation, Apache Xindice: http://xml.apache.org/xindice

[82] P. Anderson, "What is This Thing Called Configuration?", Talk at the Configuration Workshop LISA XVII, San Diego, California, USA, Oct. 2003

[83] M. Holgate, W. Partain, "The Arusha Project:A Framework for Collaborative Unix System Administration", in Proceedings of the LISA 2001, 15th Systems Administration Conference, San Diego, California, USA, Dec. 2001

[84] The World Wide Web Consortium, "RDF Primer" W3C Recommendation Feb. 2004

[85] UNIX Product Support (UPS): http://www.fnal.gov/docs/products/ups/

[86] PACMAN: http://physics.bu.edu/ youssef/pacman/

[87] S. Andreozzi, M. Sgaravatto, C. Vistoli, "Sharing a conceptual model of Grid resources and services", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, California, March 2003

[88] R. Pordes, "The Open Science Grid", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP04), Interlaken, Switzerland, Oct. 2004

[89] The GLUE schema home page: http://www.cnaf.infn.it/ sergio/datatag/glue/

[90] A. Lyon, S. Veseli, et al., "SAM-Grid Monitoring and Information Service and its Integration with MonALisa", in Proceedings of Computing in High Energy and Nuclear Physics, Interlaken, Switzerland, Oct. 2004 (to appear)

[91] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, M. Swany, "A grid monitoring architecture", technical report GWD-Perf-16-1, GGF, 2001

[92] MDS: http://www.globus.org/mds/

[93] Czajkowski, K., S. Fitzgerald, I. Foster, C. Kasselman, "Grid Information Service for Distributed Resource Sharing", in Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press, 2001

[94] X. Zhang and J. Schopf, "Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2" in Proceedings of the International Workshop on Middleware Performance (MP 2004), part of the 23rd International Performance Computing and Communications Workshop (IPCCC), April 2004.

[95] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, "Grid Information Services for Distributed Resource Sharing" in Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[96] P. Stelling, I. Foster, C. Kesselman, C.Lee, G. von Laszewski, "A Fault Detection Service for Wide Area Distributed Computations" in Proceedings of the 7th IEEE Symposium on High Performance Distributed Computing, pp. 268-278, 1998.

[97] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, S. Tuecke, "A Directory Service for Configuring High-Performance Dis-

tributed Computations" in Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing, pp. 365-375, 1997.

[98] I. Foster, G. von Laszewski, "Usage of LDAP in Globus", On line: ftp://ftp.globus.org/pub/globus/papers/ldap_in_globus.pdf

[99] The DataGrid team, "DataGrid Information and Monitoring Services Architecture: Design, Requirements and Evaluation Criteria", Technical Report, DataGrid 2002

[100] Hawkeye: http://www.cs.wisc.edu/condor/hawkeye

[101] X. Zhang, J. Freschl, and J. Schopf, "Performance Study of Monitoring and Information Services for Distributed Systems", in Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), Seattle, Washington, Jun. 2003

[102] MonALISA: http://monalisa.cacr.caltech.edu/

[103] Jini: http://www.jini.org/

[104] B. Plale, P. Dinda, G. Laszewski, "Key Concepts and Services of a Grid Information Service." ISCA 15th International Parallel and Distributed Computing Systems (PDCS), 2002

[105] V. Vaswani, P. Smith, "MySQL: The Complete Reference", published by The McGraw-Hill Companies, Aug. 2002

[106] Plale, B., "Whitepaper on Synthetic Workload for Grid Information Services/Registries", DataWorkshop 2003 held in conjunction with GlobusWorld 2003, San Diego.

[107] A.S. Rana, "A globally-distributed grid monitoring system to facilitate HPC at D0/SAM-Grid (Design, development, implementation and deployment of a prototype)", Thesis of Master in Computing Science, The University of Texas, Arlington, Nov. 2002

[108] A.S. Rana, "A globally-distributed grid monitoring system to facilitate HPC at D0/SAM-Grid (Design, development, implementation and deployment of a prototype)", Thesis of Master in Computing Science, The University of Texas, Arlington, Nov. 2002

[109] OpenLDAP: http://www.openldap.org/

[110] Yeong, W., T. Howes, and S. Kille, "Lightweight Directory Access Protocol", The Internet Engineering Task Force (IETF) RFC 1777, Mar. 1995

[111] L. Field, M. W. Schulz, "Grid Deployment Experiences: The path to a production quality LDAP based grid information system", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP04), Interlaken, Switzerland, Oct. 2004

[112] The Network Working Group, "The LDAP Data Interchange Format (LDIF)", Technical Specification, RFC 2849

[113] G. Garzoglio, I. Terekhov, A. Baranovski, S. Veseli, L. Lueking, P. Mhashilkar, V. Murthi, "The SAM-Grid Fabric services", talk at the IX International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT-03), Tsukuba, Japan; to appear in Nuclear Instruments and Methods in Physics Research, Section A

[114] SAM-Grid history plots: http://dbsmon.fnal.gov/samgrid/samgrid.html

[115] M. Satyanarayanan, "The Evolution of Coda", ACM Transactions on Computer Systems, Vol. 20, No. 2, May 2002, Pages 85-124

[116] SAM-Grid monitoring page: http://samgrid.fnal.gov:8080/

[117] PHP: http://www.php.net

[118] M.S. Neubauer, "Computing for Run II at CDF", in Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT-02), Jun 2002

[119] G.E. Graham, D. Evans, and I. Bertram, "MCRunjob: A Workflow Planner for HEP", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, California, March 2003

[120] The SAM-Grid Manual: http://www-d0.fnal.gov/computing/grid/SAMGridManual.htm

[121] Grid Job Submission Project: http://auger.jlab.org/grid/

[122] The EDG Team, "DataGrid JDL ATTRIBUTES", Internal Document of Work package 1, DataGrid-01-TEN-0142-0_2, Oct. 2003

[123] D. Thain, J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Pipeline and Batch Sharing in Grid Workloads" in Proceedings of the Twelfth IEEE Symposum on High Performance Distributed Computing

[124] G. Garzoglio, I. Terekhov, J. Snow, S. Jain, A. Nishandar, "Experience producing simulated events for the DZero experiment on the SAM-Grid", in Proceedings of Computing in High Energy and Nuclear Physics (CHEP04), Interlaken, Switzerland, Sep 2004.

[125] I. Bird, B. Hess, A. Kowalski, D. Petravick, R. Wellner, J. Gu, E. Otoo, A. Romosan, A. Sim, A. Shoshani, W. Hoschek, P. Kunszt, H. Stockinger, K. Stockinger, B. Tierney and J. Baud, "SRM (Storage Resource Manager) Joint Functional Design", Global Grid Forum Document, GGF4, Toronto, Feb. 2002

[126] MOP http://www.uscms.org/s&c/MOP/

[127] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid", in Grid Computing: Making the Global Infrastructure a Reality, published by John Wiley & Sons Inc., Dec. 2002

[128] J. Novotny, S. Tuecke, V. Welch, "An Online Credential Repository for the Grid: MyProxy", in Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press, Aug. 2001

[129] G. Kola, T. Kosar, M. Livny, "Phoenix: Making Data-Intensive Grid Applications Fault-Tolerant", Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04) pp. 251-258

[130] B.E. Chang, M. DeLap, J. Liszka, T. Murphy VII, K. Crary, R. Harper, F. Pfenning, "Towards a Functional Library for Fault-Tolerant Grid Computing.", White Paper from the Carnegie Mellon University School of Computer Science, Pittsburgh, PA, May, 2002.

[131] S.S. Tadepalli, C. Ribbens, S. Va, "GEMS: A Fault Tolerant Grid Job Management System", in proceedings of High Performance Computing Symposium 2004 (HPC2004), Arlington, VA, April 2004

[132] R. Buyya, S. Chapin, and D. DiNucci, "Architectural Models for Resource Management in the Grid", First IEEE/ACM International Workshop on Grid Computing (GRID 2000), Springer Verlag LNCS Series, Germany, Dec. 17, 2000, Bangalore, India.

[133] K. Krauter, R. Buyya, and M. Maheswaran, "A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing", International Journal of Software: Practice and Experience (SPE), Wiley Press, New York, USA, May 2002.

[134] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors", Journal of the ACM, 24(2):280-289, 1977.

[135] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems" in Proceedings of the 8th IEEE Heterogeneous Computing Workshop (HCW-99), pages 30-44, 1999.

[136] D. A. Menasce, D. Saha, S. C. D. S. Porto, V. A. F. Almeida, and S. K. Tripathi, "Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures", Journal of Parallel and Distributed Computing, 28:1-18, 1995.

[137] D. Paranhos, W. Cirne, and F. Brasileiro, "Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids, In International Conference on Parallel and Distributed Computing (Euro-Par), Lecture Notes in Computer Science, volume 2790, pages 169-180, 2003.

[138] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman, "Heuristics for scheduling parameter sweep applications in grid environments", in 9th Heterogeneous Computing Workshop (HCW), pages 349-363, 2000.

[139] J. Bruno, E.G. Jr. Coffman, and R. Sethi, "Scheduling independent tasks to reduce mean finishing-time." Comm. ACM 17, 7 (July 1974), 382-387.

[140] N. Fujimoto, K. Hagihara, "A Comparison among Grid Scheduling Algorithms for Independent Coarse-Grained Tasks", in 2004 Symposium on Applications and the Internet-Workshops (SAINT 2004 Workshops), Jan. 2004, Tokyo, Japan, p.674

[141] R. Buyya, H. Stockinger, J. Giddy, and D. Abramson, "Economic Models for Management of Resources in Peer-to-Peer and Grid Computing", In Proceedings of International Conference on Commercial Applications for High-Performance Computing, SPIE Press, August 20-24, 2001, Denver, Colorado, USA.

[142] M. Neary, A. Phipps, S. Richman, P. Cappello, "Javelin 2.0: Java-Based Parallel Computing on the Internet", in Proceedings of European Parallel Computing Conference (Euro-Par 2000), Germany, 2000.

[143] S. Chapin, J. Karpovich, and A. Grimshaw, "The Legion Resource Management System", in Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing, Apr. 1999, San Juan, Puerto Rico, Springer Verlag Press, Germany, 1999.

[144] J. Gehring and A. Streit, "Robust Resource Management for Metacomputers", in Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, USA, 2000.

[145] H. Casanova and J. Dongarra, "NetSolve: A Network Server for Solving Computational Science Problems", International Journal of Supercomputing Applications and High Performance Computing, Vol. 11, No. 3, pp 212-223, Sage Publications, USA, 1997.

[146] H. Nakada, M. Sato, S. Sekiguchi, "Design and Implementations of Ninf: towards a Global Computing Infrastructure", Future Generation Computing Systems, Metacomputing Special Issue, October 1999.

[147] N. Kapadia, R. Figueiredo, and J. Fortes, "PUNCH: Web Portal for Running Tools", IEEE Micro, May-June, 2000.

[148] R. Buyya, "Economic-based Distributed Resource Management and Scheduling for Grid Computing", Ph.D. Thesis, Monash University, Melbourne, Australia, Apr. 2002. Online at http://www.buyya.com/thesis/thesis.pdf

[149] R. Raman and M. Livny, "Matchmaking: Distributed Resource Management for High Throughput Computing", in Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, IL, Jul. 1998

[150] A. Baranovski, G. Garzoglio, I. Terekhov, A. Roy, T. Tannenbaum, "Management of Grid Jobs and Data within SAM-Grid", In Proceedings of Cluster 2004, Sept. 20-23 2004, San Diego, California (to appear)

[151] R. Raman, "Matchmaking Frameworks for Distributed Resource Management", Ph.d Dissertation, University of Wisconsin, Madison, Oct. 2000. Online: http://www.cs.wisc.edu/condor/doc/rajesh.dissert.pdf

[152] A.H. Alhusaini, V.K. Prasanna, and C.S. Raghavendra, "A Unified Resource Scheduling Framework for Heterogeneous Computing Environments". in 8th Heterogeneous Computing Workshop, (1999).

[153] X. Shi, H. Jin, W. Qiang, and D. Zou, "An Adaptive Meta-scheduler for Data-Intensive Applications", M. Li et al. (Eds.): GCC 2003, LNCS 3033, pp. 830 837, 2004.

[154] Sang-Min Park and Jai-Hoon Kim, "Chameleon: A Resource Scheduler in A Data Grid Environment" in Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 03)

[155] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, "The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid." in proceedings of Super Computing, Denver, 2000.

[156] K. Ranganathan and I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications", in Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), Edinburgh, Scotland, July 2002.

[157] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny, "Gathering at the Well: Creating Communities for Grid I/O." in Supercomputing, (Denver, CO, 2001).

[158] J. Basney, M. Livny, and P. Mazzanti, "Utilizing Widely Distributed Computational Resources Efficiently with Execution Domains.", in Computer Physics Communications, 2001.

[159] S. Vazhkudai, S. Tuecke, and I. Foster, "Replica selection in the globus data grid", in Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID), May 2001.

[160] R. Raman, M. Livny, and M. Solomon, "Resource management through multilateral matchmaking", in Proceedings of the Ninth IEEE

Symposium on High Performance Distributed Computing (HPDC), pages 290 291, Pittsburgh, PA, August 2000.

[161] H. Li, D. Groep, J. Templon, L. Wolters, "Predicting Job Start Times on Clusters", In Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid Chicago, Illinois, USA, April 19-22, 2004

[162] G. Garzoglio, S. Jain, D. Levine, A. Nishandar, I. Terekhov, "Extending the Cluster-Grid Interface Using Batch System Abstraction and Idealization", in Proceedings of Cluster Computing and Grid 2005 (CCGrid05), Cardiff, UK, May 2005

[163] S. Jain, "Abstracting the hetereogeneities of computational resources in the SAM-Grid to enable execution of high energy physics applications", Thesis of Master in Computing Science, The University of Texas, Arlington, Dec. 2004

[164] A. Nishandar, "Grid-Fabric Interface For Job Management In Sam-Grid, A Distributed Data Handling And Job Management System For High Energy Physics Experiments", Thesis of Master in Computing Science, The University of Texas, Arlington, Dec. 2004

[165] I. Bird (LCG Deployment Area Manager & EGEE Operations Manager), "Operating the LCG and EGEE Production Grids for HEP", plenary talk at Computing in High Energy and Nuclear Physics (CHEP04), Interlaken, Switzerland, Oct. 2004

[166] Julien Devémy, "BQS and the Grid: problems and solutions", talk at HEPiX 2005, Workshop on Batch systems, Karlsruhe, Germany, May 2005

[167] Bernard Chambon, "BQS at CCIN2P3", talk at HEPiX 2005, Workshop on Batch systems, Karlsruhe, Germany, May 2005

[168] G. Garzoglio, S. Jain, D. Levine, A. Nishandar, I. Terekhov, "Black Hole Effect: Detection and Mitigation of Application Failures due

to Incompatible Execution Environment in Computational Grids", in Proceedings of Cluster Computing and Grid 2005 (CCGrid05), Cardiff, UK, May 2005

[169] FCP - Farm Remote Copy Utility: http://www-isd.fnal.gov/fcp/

[170] Disk Farm project: http://www-isd.fnal.gov/dfarm

[171] RTE project: http://www-d0.fnal.gov/~ritchie/CPBdemo.html

[172] Globus Access to Secondary Storage (GASS) server documentation page: http://www.globus.org/toolkit/docs/2.4/gass/

[173] Czajkowski, K., D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe, "The WS-Resource Framework", 2004. www.globus.org/wsrf/

[174] Personal Communication, Todd Tannenbaum and Zach Miller, Department of Computer Sciences, University of Wisconsin, Madison

[175] A. Rajendra, "Integration of the SAM-Grid Infrastructure to the DZero Data Reprocessing Effort", Thesis of Master in Computing Science, The University of Texas, Arlington, Dec. 2005